27
SERIA
PC





LIMBAJELE C SI C++ PENTRU INCEPATORI

VOLUMUL I

LIMBAJUL C



LIMBAJELE C SI C++ PENTRU INCEPATORI

LIMBAJUL C

REEDITARE

VOLUMUL I

CLUJ-NAPOCA 1995

Autor
Liviu Negrescu

A confruntat cu originalul
Irina Mitrov

Editura Albastră



Coordonator serie
Smaranda Derveşteanu

Tehnoredactare computerizată
Codruţa Poenaru

Coperta
Liviu Derveşteanu

Tipărit

Comanda nr. 123
Imprimeria Ardealul Cluj



Tiraj 2000 exemplare

CUPRINS

PREFAŢĂ

Deceniul anilor '90 este deceniul consacrat *programării orientate spre obiecte* (în engleză Object Oriented Programming sau prescurtat OOP) la fel cum anii '70 au fost numiţi anii *programării structurate*.

Programarea orientată spre obiecte este un *stil* de programare care permite abordarea eficientă a aplicaţiilor complexe. Acesta se asigură pe baza:

- elaborării de componente reutilizabile, extensibile şi uşor modificabile, fără a reprograma totul de la început;
- construirii de biblioteci de module extensibile;
- implementării simple a interacţiunilor programelor cu mediul de calcul prin utilizarea unor elemente standardizate.

Pe scurt, putem afirma că, programarea orientată spre obiecte permite să se implementeze sisteme complexe cu ajutorul unor componente individuale, standardizate, reutilizabile, extensibile şi uşor modificabile.

Această concepţie a influenţat dezvoltarea limbajelor de programare. Ea s-a manifestat începînd din anii '60 cînd s-a introdus conceptul de *obiect* în limbajul *Simula*.

În principiu, un obiect are o structură de date bine precizată şi totodată sînt definite operaţiile care pot utiliza componentele lui.

Simula se consideră că se află la baza limbajelor care permit programarea orientată spre obiecte.

În cele ce urmează vom spune despre un limbaj de programare că este *orientat spre obiecte* dacă permite programarea orientată spre obiecte.

În deceniul anilor '70, programarea orientată spre obiecte s-a utilizat mai ales prin intermediul limbajului *Smalltalk*. Aşa cum s-a amintit mai sus, în acest deceniu s-au pus bazele programării structurate care a avut un succes deosebit prin utilizarea limbajului *Pascal*.

În anul 1972 a apărut *limbajul C*. Acesta este un limbaj de programare care are o destinaţie universală. Autorii acestui limbaj sînt Dennis M. Ritchie şi Brian W. Kernighan de la Bell Laboratories.

Limbajul C a fost proiectat în ideea de a asigura implementarea portabilă a sistemului de operare UNIX. Un rezultat direct al acestui fapt este acela că programele scrise în limbajul C au o *portabilitate* foarte bună.

În mod intuitiv, spunem că un program este portabil dacă el poate fi

transferat uşor de la un tip de calculator la altul. În principiu, toate limbajele de nivel înalt asigură o anumită portabilitate a programelor. La ora actuală se afirmă că programele scrise în C sînt cele mai portabile.

Prin anii '80 interesul pentru programarea orientată spre obiecte a crescut, ceea ce a condus la apariţia de limbaje care să permită utilizarea ei în scrierea programelor. Limbajul C a fost şi el dezvoltat în această direcţie. Astfel, în anul 1980 s-a dat publicităţii *limbajul C++*.

Limbajul C++ a fost elaborat de Bjarne Stroustrup de la AT&T. El este un superset al limbajului C şi permite utilizarea principalelor concepte ale programării orientate spre obiecte.

Limbajul Pascal a fost şi el dezvoltat în această direcţie.

La ora actuală, majoritatea limbajelor de programare moderne au fost dezvoltate în direcţia programării orientate spre obiecte.

În prezent, anii '80 sînt consideraţi ca fiind decada care a lansat programarea orientată spre obiecte.

Limbajul C++, ca şi limbajul C, se bucură de o portabilitate mare şi este implementat pe o gamă largă de calculatoare începînd cu microcalculatoare şi pînă la cele mai mari supercalculatoare. Între diferitele implementări ale limbajului C++ există diferenţe legate de specificul calculatoarelor, dar aceste diferenţe nu sînt esenţiale.

Limbajul C++ a fost implementat pe microcalculatoarele compatibile IBM PC în mai multe variante. Cele mai importante implementări ale limbajului C++ pe aceste calculatoare sînt cele realizate de firmele Microsoft şi Borland. Aceste firme au implementat limbajul C++ pe microcalculatoarele respective în mai multe versiuni.

În cartea de faţă se prezintă facilităţile de bază ale limbajului C++ care sînt comune diferitelor implementări ale sale. Exerciţiile prezentate în cartea de faţă au fost rulate folosind compilatorul TURBO C++ v2.0 (acest compilator este marcă înregistrată a firmei Borland).

Conceptele programării orientate spre obiecte au influenţat în mare măsură dezvoltarea limbajelor de programare din ultimii ani. De obicei, multe limbaje au fost extinse aşa încît ele să admită conceptele mai importante ale programării orientate spre obiecte. Uneori s-au făcut chiar mai multe extensii ale aceluiaşi limbaj.

De exemplu, în prezent asistăm la extensiile limbajuluiC++. O astfel de extensie o constitue *limbajul E* al cărui autori sînt Joel E. Richardson, Michael J. Carey şi Daniel T. Schuh de la universitatea Wisconsin Madison.

Acest limbaj a fost proiectat pentru a permite exprimarea simplă a

tipurilor şi operaţiilor interne sistemelor de gestiune a bazelor de date.

Printre altele, limbajul E permite crearea şi gestiunea *obiectelor persistente*, lucru deosebit de important pentru sistemele de gestiune a bazelor de date.

O altă extensie a limbajului C++ este *limbajul O* dezvoltat la Bell Laboratories. Cele două limbaje (E şi O) sînt în esenţă echivalente.

Despre limbajul E se afirmă că este în principal orientat spre scrierea programelor sistem, în particular pentru implementarea sistemelor de gestiune a bazelor de date.

Despre limbajul O se spune că încearcă să îmbine facilităţile de nivel înalt cu cele ale programării sistem.

O altă extensie a lui C++ este limbajul Avalon/C++. Ea a fost realizată de Detlefs D., Herlihy M. şi Wing J. Acest limbaj este destinat pentru a suporta calcul distribuit.

Printre altele, aceste ultime două extensii admit şi ele obiecte persistente.

În prezent, conceptele programării orientate spre obiecte au aplicaţii importante nu numai în ce priveşte dezvoltarea limbajelor de programare, ci şi la implementarea altor sisteme complexe.

Cele mai mari realizări obţinute pînă în momentul de faţă sînt legate de implementarea *sistemelor de gestiune a bazelor de date* şi a *interfeţelor utilizator*.

Limbajul E a fost utilizat, de către autorii lui, la implementarea proiectului EXODUS care este un sistem de gestiune a unei baze de date "*extensibile*".

În prezent există numeroase proiecte de acest fel implementate cu ajutorul conceptelor programării orientate spre obiecte. Bazele de date respective se numesc *baze de date orientate spre obiecte*.

Interfeţele utilizator au atins o dezvoltare mare datorită facilităţilor oferite de componentele hardware ale diferitelor tipuri de calculatoare. În principiu ele simplifică interacţiunea dintre programe şi utilizatorii acestora. Astfel, diferite comenzi, date de intrare sau rezultate pot fi exprimate simplu şi natural utilizînd diferite standarde care conţin ferestre, bare de meniuri, cutii de dialoguri, butoane etc. Un dispozitiv primar de selectare utilizat foarte frecvent şi cu mult succes este aşa numitul "*şoarece*" (mouse). Toate acestea conduc la interfeţe simple şi vizuale, accesibile pentru utilizatorii finali.

Implementarea interfeţelor este mult simplificată prin utilizarea limbajelor orientate spre obiecte. Aceasta mai ales datorită posibilităţilor de a utiliza componente standardizate aflate în biblioteci specifice.

Importanţa aplicării conceptului de reutilizare prin intermediul limbajelor orientate spre obiecte rezultă din faptul că interfeţele utilizator adesea ocupă 40% din codul total al aplicaţiei.

Limbajele utilizate frecvent la implementarea interfeţelor utilizator sînt limbajele C++ şi Pascal.

Firma Borland comercializează o bibliotecă de componente standardizate care pot fi utilizate la implementarea interfeţelor utilizator folosind unul din limbajele C++ şi Pascal.

Produsul respectiv se numeşte *Turbo Vision*. Utilizarea lui pentru implementarea de interfeţe utilizator presupune cunoaşterea a cel puţin unuia din aceste limbaje de programare.

Interfeţele utilizator implementate cu ajutorul limbajelor orientate spre obiecte le vom numi *interfeţe utilizator orientate spre obiecte*.

De obicei, interfeţele utilizator gestionează ecranul în mod grafic. O astfel de interfaţă utilizator se numeşte *interfaţă utilizator grafică*.

Una dintre cele mai populare interfeţe utilizator grafice pentru calculatoarele compatibile IBM PC este produsul *Windows* oferit de firma Microsoft. Produsul a fost lansat în noiembrie 1985 şi pînă acum a fost vîndut în cîteva milioane de exemplare. În prezent este lansată în exploatare versiunea v3.1.

Windows este un mediu de programare care amplifică facilităţile sistemului de operare MS-DOS. Aplicaţiile Windows se pot dezvolta folosind diferite medii de dezvoltare ca: Turbo C++ pentru Windows, Pascal pentru Windows, Microsoft C++ 7.0, Microsoft Visual Basic, Visual C şi Visual C++.

Componentele Visuale permit specificarea în mod grafic a interfeţei utilizator, a unei aplicaţii, folosind şoarecele, iar aplicaţia propriu-zisă se programează într-un limbaj de tip Basic, C sau C++.

Din cele de mai sus, putem afirma că limbajul C++ se bucură de o popularitate crescută fiind utilizat în măsură mare la implementarea de aplicaţii complexe.

Dacă prin anii '70 se considera că o singură persoană este rezonabil să se ocupe cu programe de pînă la 4-5 mii de instrucţiuni, în prezent, în condiţiile folosirii limbajelor de programare orientate spre obiecte, această medie este de 25 mii de instrucţiuni.

Cartea de faţă se adresează tuturor celor care doresc să se familiarizeze cu limbajul C++. Din motive de spaţiu, în carte nu sînt date detaliile cu privire la noţiunile de bază ale programării, arhitectura sau modul de funcţionare al unui calculator, considerîndu-se că cititorul este familiarizat cu cel puţin un limbaj de programare. Tot din lipsă de spaţiu nu este descris mediul de dezvoltare integrat TURBO C++ el fiind foarte asemănător cu mediile integrate de dezvoltare ale celorlalte compilatoare furnizate de firma Borland. Cititorul care nu este familiarizat cu astfel de medii poate consulta alte cărţi care descriu astfel de medii (vezi de exemplu [17]).

În încheiere exprim mulţumiri D-nei Irina Mitrov pentru observaţiile şi sugestiile care mi le-a făcut în legătură cu conţinutul cărţii şi al exerciţiilor, precum şi cu privire la îmbunătăţirea clarităţii şi exprimării diferitelor noţiuni. Menţionez activitatea intensă depusă de D-na Irina Mitrov şi fiica mea, Lavinia Negrescu, în legătură cu munca de introducere pe calculator a textului cărţii. Tot pe această cale doresc să-i mulţumesc fiului meu, Dan Negrescu, pentru ajutorul pe care mi l-a dat în legătură cu formularea şi rezolvarea unor exerciţii care sînt legate de facilităţile sistemului de operare MS-DOS.

De asemenea, mulţumesc editurii **Casa de Editură Albastră** pentru efortul depus în vederea publicării îngrijite a cărţii de faţă.

Autorul

INTRODUCERE

Limbajul C++ este un superset al limbajului C avînd o utilizare universală şi care permite scrierea de programe orientate spre obiecte.

Faptul că limbajul C++ este un superset al limbajului C înseamnă că orice program scris în limabjul C este în acelaşi timp şi un program scris în limbajul C++. Această afirmaţie este adevărată cu mici excepţii, de exemplu,compilatorul C++ face unele controale suplimentare faţă de compilatorul C. Aceste controale se referă în primul rînd la tipurile parametrilor funcţiilor, p ecum şi la tipurile valorilor returnate de ele.

Faptul că limbajul C++ permite scrierea de programe orientate spre obiecte înseamnă că el permite utilizarea unor concepte de bază ale acestui stil de programare. Acest fapt reprezintă un salt calitativ faţă de limbajul C, ceea ce se exprimă sugestiv şi prin numele limbajului (C++). El nu a fost numit limbajul D deoarece este o extensie a lui C.

Ţinînd seama că operatorul "++", din limbajul C, reprezintă operaţia de incrementare, numele C++ se poate citi "C incrementat".

În principiu, putem afirma că limbajul C++ este un "C mai bun", care oferă atît facilităţile limbajului C cît şi programarea orientată spre obiecte.

Facilităţile principale ale limbajului C sînt:

- portabilitatea mare a programelor;
- flexibilitatea în programare;
- programe compacte;
- lucrul pe biţi;
- calcul de adrese.

Menţionăm că limbajul C iniţial a fost proiectat în ideea de a asigura o portabilitate bună a programelor.

Ca o consecinţă a acestui deziderat amintim lipsa, din limbajul C, a instrucţiunilor de intrare/ieşire. Aceste operaţii se realizează prin funcţii de bibliotecă. Eliminarea din limbaj a instrucţiunilor de intrare/ieşire rezultă din faptul că operaţiile de intrare/ieşire sînt dependente de particularităţile hardware ale calculatoarelor.

Flexibilitatea în programare rezultă, printre altele, din numărul mai redus de *controale* pe care îl face compilatorul C, faţă de alte compilatoare, de exemplu faţă de compilatorul limbajului Pascal. Acest fapt a provocat o serie de discuţii, întrucît absenţa unor controale severe la compilare se

consideră că îngreunează depistarea unor erori în programare. Cu toate acestea, flexibilitatea în programare obţinută în acest fel s-a dovedit a fi adesea utilă şi trebuie considerată o facilitate în plus a limbajului decît o parte negativă a lui.

Alte facilităţi care măresc flexibilitatea în progaramare sînt legate de prezenţa în limbaj a unor instrucţiuni care asigură revenirea simplă dintr-o funcţie, ieşirea din cicluri etc.

Compilatorul C++ realizează unele controale suplimentare faţă de compilatorul C. Cu toate acestea programarea în limbajul C++ se bucură de o flexibilitate mare.

O altă facilitate importantă a limbajului C este posibilitatea de a compacta programele sursă. Limbajul C permite o compactare bună a programelor sursă pe seama unor construcţii specifice, cum sînt: expresia de atribuire, expresia condiţională, operatorii de incrementare şi decrementare etc.

Limbajul C oferă facilităţi specifice limbajelor de asamblare, cum sînt lucrul pe biţi şi calculul cu adrese. Acest fapt permite adesea scrierea de programe optime, atît în ceea ce priveşte memoria, cît şi timpul de execuţie.

Limbajul C se consideră că este un intermediar între limbajele de nivel înalt şi cele de asamblare. El a cîştigat o popularitate mare în decursul anilor datorită facilităţilor amintite mai sus.

La ora actuală există un standard ANSI al limbajului C, dar numeroasele sale implementări se abat de la acest standard.

În cartea de faţă ne vom referi la versiunea TURBO C v2.0 care este marcă înregistrată a firmei BORLAND şi este implementată pe calculatoarele compatibile IBM PC.

Compilatorul TURBO C poate fi apelat prin intermediul *mediului integrat de dezvoltare* (Integrated Development Environment) sau printr-o linie de comandă.

Fără a intra în detalii cu privire la mediul integrat de dezvoltare TURBO C, indicăm mai jos o secvenţă simplă de comenzi pentru a lansa în execuţie un program C nou:

- după setarea pe directorul corespunzător se tastează:
  - tc;
  - <Alt>-F;
  - N.
- se editează programul sursă folosind editorul mediului TURBO C;

- se acţionează tasta F2;
- se indică numele fişierului pentru salvarea programului sursă pe disc;
- se realizează compilarea, link-editarea şi lansarea în execuţie a programului tastînd:

  <Ctrl>-F9
- pentru a vizualiza rezultatele execuţiei programului se tastează:

  <Alt>-F5
- se revine în fereastra de editare a mediului acţionînd o tastă oarecare.

Tasta F1 poate fi acţionată pentru a obţine informaţii suplimentare. Se revine în faza precedentă acţionînd tasta ESC.

Pentru a ieşi din mediul integrat de dezvoltare TURBO C se poate tasta:

<Alt>-X

Compilatorul TURBO C++ poate fi şi el apelat prin intermediul mediului integrat de dezvoltare TURBO C++ sau printr-o linie de comandă.

Mediul integrat de dezvoltare TURBO C++ este asemănător cu cel al compilatorului TURBO C. De exemplu, secvenţa de mai sus poate fi utilizată şi în cazul compilatorului TURBO C++ v2.0 cu singura diferenţă că în loc de *tc* se va tasta *bc*. Menţionăm că fişierele cu programe sursă C au extensia .C, iar cele cu programele sursă C++ au extensia .CPP. Programele sursă C din această carte au fost compilate cu compilatorul C++ şi ele au extensia .CPP.

Limbajul C, alături de limbajul Pascal şi celelalte limbaje precedente lor sînt adecvate *programării procedurale*.

Programarea procedurală este un model de programare utilizat frecvent încă din faza iniţială de existenţă a limbajelor de programare.

La baza acestei programări se află *procedura*. Aceasta poate fi considerată ca un proces de *abstractizare*. Acest proces se realizează pe baza *parametrilor* care permit realizarea procedurii făcînd *abstracţie de valorile concrete* pentru care să se execute procedura.

De exemplu, la realizarea unei proceduri pentru calculul funcţiei *sin* se face abstracţie de valoarea concretă a unghiului. Procedura are un parametru care ne permite să neglijăm valoarea concretă a unghiului. Procedura se defineşte în aşa fel încît ea să calculeze valoarea funcţiei *sin* în

funcţie de valoarea parametrului ei, valoare care se concretizează la execuţie în momentul apelului procedurii respective.

Acest proces de abstractizare prin intermediul parametrilor procedurilor se numeşte *abstractizare procedurală.*

Un alt aspect al acestui proces de abstractizare este acela că procedura poate fi privită ca o "cutie neagră". La apelul ei nu se au în vedere detaliile de realizare ale procedurii. De exemplu, la apelul procedurii de calcul a valorii funcţiei *sin* sînt neglijate detaliile cu privire la algoritmul de calcul al funcţiei respective. La această fază interesează "tipul" parametrului şi eventual precizia de calcul a funcţiei respective.

Abstractizarea procedurală a fost folosită pe scară largă la rezolvarea multor probleme aplicative sau de sistem. Importanţa ei creşte odată cu creşterea complexităţii problemei de rezolvat. De obicei, problemele complexe se descompun în subprobleme mai simple. Acestea, la rîndul lor, pot fi şi ele descompuse în continuare şi aşa mai departe, pînă cînd se ajunge la componente suficient de simple. Acest proces de descompunere se realizează printr-un proces de abstractizare care permite neglijarea diferitelor detalii. În felul acesta pot fi puse în evidenţă procese de calcul care prin utilizări de parametri pot fi exprimate printr-o procedură.

Pasul următor programării procedurale a fost *programarea modulară.* Deşi nu există o definiţie riguroasă pentru noţiunea de modul, citîndu-l pe Bjarne Stroustrup, se poate afirma că *modulul* este un set de proceduri înrudite împreună cu datele pe care le manevrează (vezi[14]).

Ceea ce trebuie să mai precizăm este faptul că datele manevrate de aceste proceduri, care compun un modul, sînt de obicei "ascunse" în modulul respectiv. Aceasta înseamnă că în afara modulului nu se are acces direct la datele ascunse de modul. Se are acces la ele numai indirect, prin intermediul procedurilor modulului respectiv. Limitarea accesului la date prin ascunderea lor într-un modul conduce la o protejare a lor, ceea ce prezintă importanţă mai ales în cazul programelor complexe.

Un exemplu simplu de modul este modulul pentru implementarea unei *stive*. În forma cea mai simplă, stiva poate fi considerată ca fiind o zonă de memorie în care pot fi păstrate date, de obicei de un acelaşi tip. Datele pot fi scoase din stivă numai în ordinea inversă păstrării lor. Ultima dată pusă în stivă se spune că este în vîrful ei. Data scoasă din stivă este totdeauna aceea aflată în vîrful stivei. Totodată, după scoaterea unei date din stivă, în vîrful stivei se va afla data care a fost pusă pe stivă imediat înaintea celei scoase din stivă. În felul acesta, totdeauna se scoate din stivă ultima dată păstrată (pusă) în stivă. Se obişnuieşte să se spună că stiva este o zonă de memorie gestionată după principiul: ultimul pus pe stivă este primul scos din stivă

(last in-first out sau pe scurt LIFO). Este important ca acest principiu să fie respectat. De exemplu, nu este admis să se scoată din stivă sau să se aibă acces la un element care nu este în vîrful stivei. De asemenea, cînd se pune un element pe stivă, acesta poate fi pus numai după cel aflat în vîrful stivei şi prin aceasta el devine elementul *din* vîrful stivei. Din această cauză, zona de memorie prin care se implementează stiva trebuie "ascunsă", accesul la ea realizîndu-se numai prin intermediul a două proceduri:

- una care să pună o dată pe stivă (procedură care de obicei se numeşte *push*);

şi

- una care să scoată o dată din stivă (procedură care de obicei se numeşte *pop*).

De obicei, la aceste proceduri se mai adaugă o procedură pentru iniţializarea stivei. Această procedură are drept scop vidarea stivei. În urma apelului ei, stiva devine vidă (această procedură de obicei se numeşte *clear*).

Cele trei proceduri amintite mai sus se consideră "înrudite" şi împreună cu zona de memorie utilizată pentru implementarea stivei se grupează într-un modul. Utilizatorul nu poate gestiona direct această zonă de memorie decît numai prin intermediul celor 3 proceduri amintite mai sus.

Un modul similar se poate defini pentru a gestiona o zonă de memorie conform principiului: primul pus în zona respectivă este şi primul scos (first in-first out sau pe scurt FIFO). O astfel de zonă de memorie se spune că formează o *coadă*.

Adesea, ascunderea datelor în modul se extinde şi asupra procedurilor, limitîndu-se accesul din afara modulului la unele proceduri din compunerea lui.

Programarea modulară corespunde mai bine ideii amintite mai sus cu privire la descompunerea problemelor complexe în subprobleme mai simple. Ea permite un grad mai înalt de abstractizare pe seama ascunderii datelor şi a procedurilor.

Limbajul C a fost proiectat în aşa fel încît să permită realizarea de module cu ascunderea datelor şi a procedurilor. Astfel, datele şi procedurile declarate cu *static* au o valabilitate limitată în cadrul fişierului sursă în care sînt declarate.

Programarea modulară are anumite limite. Aşa de exemplu, dacă este nevoie de mai multe stive, atunci programarea modulară este o soluţie destul de complexă pentru a gestiona stivele respective.

O rezolvare mai simplă şi naturală s-ar obţine dacă am putea defini date de tip stivă, aşa cum putem defini, de exemplu, date de tip întreg, flotant sau caracter.

Toate limbajele de nivel înalt pun la dispoziţia utilizatorului un anumit număr de tipuri predefinite. De exemplu, limbajul FORTRAN are tipurile predefinite *integer* şi *real*. Limbajul C are şi el aceste tipuri predefinite (întreg şi flotant) şi în plus mai pune la dispoziţia utilizatorului tipul caracter.

Un *tip de date* descrie un set de date pe care le mai numim şi obiecte, care au aceeaşi reprezentare. De asemenea, există un număr de operaţii asociat cu un tip de date. De exemplu, cu tipul întreg se asociază cele patru operaţii aritmetice şi eventual şi altele.

Ulterior s-a constatat că este util ca limbajul să permită utilizatorului să definească el însuşi tipuri diferite de cele predefinite în limbaj. În acest scop, în limbajul Pascal s-a introdus noţiunea de *înregistrare* (record) care permite definiri de grupe de date care nu neapărat au un acelaşi tip. O astfel de grupă de date reprezintă un tip nou de date. O facilitate analogă există şi în limbajul C. În acest caz se utilizează noţiunea de *structură* (struct). Atît construcţia record din Pascal cît şi struct din limbajul C definesc *reprezentarea* tipului nou de dată.

Utilizatorul defineşte, de asemenea, operaţii cu date de acest tip prin intermediul unor proceduri sau funcţii.

Tipurile definite în acest fel se numesc *tipuri definite de utilizator* sau mai scurt *tipuri utilizator*. Un exemplu de tip utilizator folosit frecvent poate fi tipul de număr complex. Acesta are o reprezentare care se defineşte simplu în limbajulC folosind construcţia struct. Numărul complex este o grupă formată din două numere flotante, primul reprezintă partea reală a numărului complex, iar celălalt partea imaginară a lui.

În continuare, se pot defini variabile care să aibă tipul complex, exact la fel cum se definesc variabile de tipul întreg, flotant sau caracter.

Pentru a realiza operaţii cu numere complexe, utilizatorul defineşte funcţii corespunzătoare. Operaţiile care se execută asupra datelor care au tipuri predefinite au diferite facilităţi care lipsesc în cazul tipurilor utilizator. Aşa de exemplu, pentru datele de tip întreg sau flotant se utilizează operatorii obişnuiţi pentru operaţiile aritmetice (+, -, * şi /). Acest lucru nu este posibil şi pentru numerele de tip complex. De asemenea, o serie de controale şi conversii se realizează automat în cazul utilizării datelor de tipuri predefinite.

Un alt neajuns al tipurilor utilizator, definite ca mai sus, este faptul că nu

se asigură nici o protecţie asupra componentelor unei date.

Aceste neajunsuri rezultă mai ales din faptul că la definirea tipurilor utilizator de fapt se definesc numai reprezentările datelor respective. Nu se precizează nicăieri operaţiile posibile asupra datelor respective. Chiar dacă astfel de operaţii sînt definite prin proceduri sau funcţii, compilatoarele nu au informaţii în acest sens. De aceea, pasul următor constă în posibilitatea de a preciza în limbaj procedurile sau funcţiile care au acces la componentele unei date de tip utilizator. O astfel de extensie s-a realizat foarte simplu, de exemplu în cazul construcţiei struct din limbajul C, alături de definirea reprezentării componentelor se indică şi lista funcţiilor care definesc operaţiile asupra lor. Aceasta însă, nu înseamnă că numai funcţiile respective au acces la datele componente ale structurii. Cu alte cuvinte, prin enumerarea funcţiilor în cadrul construcţiei struct nu se oferă nici o protecţie a componentelor structurii respective, ci numai se indică faptul că datele componente ale structurii sînt prelucrate prin funcţiile enumerate, dar la ele putem avea acces şi prin alte funcţii.

Mai tîrziu s-a constatat că adesea anumite componente date şi eventual şi componente funcţii ataşate unui tip definit de utilizator este bine să fie protejate pentru a înlătura accesul neautorizat. În felul acesta s-a ajuns la noţiunea de *tip abstract de dată*. Această noţiune depăşeşte posibilităţile limbajului C. Tipurile abstracte de date pot fi definite în limbajul C++ folosind noţiunea de *clasă*. O clasă, deci, defineşte atît reprezentarea datelor tipului respectiv cît şi funcţiile care au acces şi pot prelucra datele respective.

La definirea unei clase se indică componentele dată şi funcţie la care au acces utilizatorii clasei (componente *publice*). Componentele care nu sînt publice permit un acces limitat şi în felul acesta ele sînt protejate faţă de accesurile neautorizate. Se obişnuieşte să se spună că, clasele "încapsulează" datele. Acest proces de încapsulare este deosebit de important permiţînd definirea de componente *protejate* şi *reutilizabile*. Deosebirea dintre clasă şi construcţia struct constă chiar în ideea de protecţie. Un tip utilizator definit printr-o construcţie struct poate fi definit printr-o clasă care are toate componentele dată şi funcţie, publice.

O dată de un tip abstract (adică tip definit printr-o clasă) se spune că este un *obiect*. De asemenea, se obişnuieşte să se spună că obiectul este o *instanţiere* a clasei care defineşte tipul său.

Reluînd exemplul cu privire la definirea şi gestiunea stivelor, observăm că de data aceasta putem introduce tipul abstract de dată *stiva* prin intermediul unei clase. Zona de memorie care se organizează ca stivă este o componentă dată protejată a clasei respective. O altă componentă dată protejată este cea care defineşte vîrful stivei. Componentele funcţie,

publice, ale clasei sînt funcţiile *push*, *pop* şi *clear*. Ele pot fi apelate din diferite funcţii ale programului. Deoarece componentele date sînt protejate, la ele nu se are acces decît prin intermediul celor trei funcţii publice indicate mai sus. Prin instanţierea acestei clase se obţine un obiect de tip stivă. În felul acesta se pot obţine atîtea obiecte de tip stivă, cîte sînt necesare, exact la fel cum se pot defini atîtea date de un tip predefinit, cîte sînt necesare.

În mod analog, se poate defini, printr-o clasă, tipul abstract de dată pentru numere complexe. În acest caz sînt necesare două componente dată, una pentru partea reală şi una pentru partea imaginară. Ambele componente sînt protejate. Cele patru operaţii aritmetice asupra numerelor complexe se definesc prin patru componente funcţie care sînt publice.

Un număr complex este o instanţiere a acestei clase. Se pot instanţia atîtea numere complexe cîte sînt necesare.

Mai mult decît atît, componentele funcţie care definesc cele patru operaţii aritmetice cu numere de tip complex pot fi definite în aşa fel încît operatorii acestor operaţii să poată fi folosiţi nu numai pentru date de tipuri predefinite ale limbajului, ci şi pentru datele de tip complex. Aşa de exemplu, funcţia pentru adunarea a două numere complexe poate fi definită în aşa fel ca expresia x+y să fie valabilă nu numai pentru cazurile în care x şi y sînt de tip întreg sau flotant, ci şi cînd ei sînt de tip complex.

Cu alte cuvinte, este posibil ca valabilitatea de aplicare a unui operator al limbajului să fie extinsă şi pentru tipuri abstracte de date. Această posibilitate trebuie privită ca un mijloc de extindere a valabilităţii operatorilor existenţi. Acest mijloc îl vom numi în continuare *supraîncărcarea* operatorilor.

Menţionăm că, clasele au de obicei şi alte componente funcţie care permit realizarea unor operaţii specifice tipurilor abstracte de date. De exemplu, obiectele pot fi iniţializate la instanţierea lor printr-o componentă funcţie specială, care se numeşte *constructor*.

De asemenea, o altă componentă funcţie specială realizează "distrugerea" obiectului. Ea se numeşte *destructor*.

Alte componente funcţie se definesc pentru a copia obiecte, a face diferite conversii etc.

Toate aceste facilităţi apropie mult comportamentul obiectelor instanţiate prin clase de cele care au tipuri predefinite.

Programarea care utilizează tipuri abstracte de date este un stil de programare superior programării modulare. Un astfel de stil de programare se numeşte *programare* prin *abstractizarea datelor*. În esenţă, ea constă în definirea de tipuri abstracte de date pentru fiecare *concept* necesar la

rezolvarea unei probleme concrete, concept care nu este predefinit în limbajul de programare utilizat (mai sus s-au indicat conceptele de stivă şi număr complex). În felul acesta programarea devine mai simplă şi mai naturală.

De aici şi pînă la stilul programării orientate spre obiecte mai este un singur pas.

De obicei, diferite tipuri abstracte de date au elemente comune. Precizarea lor conduce la o "ierarhizare" a tipurilor abstracte de date. În vîrful ierarhiei se află clasa ce conţine elementele comune celorlalte clase. Ea este clasa cea mai generală. Pe nivelul următor al ierarhiei se află alte clase care conţin elementele clasei din vîrful ierarhiei, precum şi alte componente specifice tipurilor abstracte pe care le definesc. În general, dacă o clasă conţine şi alte componente decît cele care sînt comune pentru una sau mai multe clase, atunci aceasta se află la un nivel mai inferior al ierarhiei decît clasa care defineşte elementele comune lor.

Ierarhizarea claselor are o importanţă mare, deoarece atributele unei clase rămîn valabile pentru clasele de pe nivelul următor ei din ierarhie. Această proprietate se numeşte *moştenire*.

Pe scurt, moştenirea permite definirea de tipuri abstracte de date noi prin adăugarea de componente dată şi/sau funcţie la un tip abstract de date deja existent. Prin aceasta se pot elabora simplu componente extensibile.

O clasă care se defineşte adăugînd componente noi unei clase date se numeşte *clasă derivată*, iar clasa compusă din elementele comune (la care s-au adăugat componentele noi) se numeşte *clasă de bază*. Această terminologie este utilizată în C++.

Uneori clasa derivată se mai numeşte şi subclasă a clasei de bază, iar clasa de bază se numeşte superclasă a unei clase derivate a ei.

Noţiunea de moştenire se aplică cu succes la prelucrarea conceptelor ierarhice din lumea reală.

Modelarea ierarhiilor din lumea reală conduce la o ierarhie de tipuri abstracte de date care în limbajul C++ se defineşte printr-o ierarhie de clase bazată pe proprietatea de moştenire.

Un exemplu simplu de ierarhie din lumea reală este o comunitate de persoane împreună cu conducătorii acestei comunităţi. În acest exemplu simplu, există două concepte şi anume conceptul *persoană* şi conceptul de *conducător*.

Conceptul de persoană se modelează printr-o clasă care conţine datele unei persoane:

- nume;
- prenume;
- data naşterii;
- localitate;
- domiciliu;
- profesiune;
- studii;
- stare civilă;
- număr copii;
- încadrare;
- vechime în muncă;
- salarizare;
- cod.

Conceptul de conducător derivă din cel de persoană deoarece orice conducător are toate atributele specifice unei persoane. De aceea, conceptul de conducător se modelează printr-o clasă care derivă din clasa persoană.

Conceptul de conducător presupune elemente noi, cum ar fi de exemplu cunoştinţe despre conducere, specializare, funcţie etc.

Un alt exemplu simplu este ales din geometrie şi se referă la patrulatere.

Dacă se consideră clasa care defineşte conceptul de paralelogram (patrulater cu laturile opuse paralele), atunci conceptul de dreptunghi poate fi modelat printr-o clasă derivată din cea a paralelogramului. Într-adevăr, dreptunghiul este un paralelogram la care mai adăugăm condiţia ca să aibă un unghi drept. Deci clasa dreptunghi este o clasă derivată a clasei paralelogram, iar aceasta din urmă este o clasă de bază a clasei dreptunghi. Ierarhia poate fi continuată cu pătratul. Într- adevăr, pătratul este un dreptunghi cu toate laturile egale, deci clasa corespunzătoare pătratului este derivată din cea a dreptunghiului. De asemenea, clasa pentru dreptunghi este clasă de bază pentru clasa corespunzătoare pătratului.

Deoarece rombul este un paralelogram cu toate laturile egale, clasa corespunzătoare acestuia este şi ea o clasă derivată din clasa corespunzătoare paralelogramului. Deci, clasa pentru paralelogram este clasă de bază atît pentru clasa corespunzătoare dreptunghiului cît şi pentru cea corespunzătoare rombului.

La rîndul ei, clasa corespunzătoare paralelogramului poate fi şi ea derivată dintr-o clasă mai generală corespunzătoare patrulaterelor.

Numărul exemplelor poate fi mărit uşor, deoarece lumea reală este plină de ierarhii. De aici decurge importanţa mare a conceptului de moştenire care distanţează substanţial stilul programării orientate spre obiecte de cel al programării prin abstractizarea datelor.

În concluzie, programarea procedurală este cea mai veche şi ea se află la nivelul cel mai de jos din punctul de vedere al stilului de programare. La nivelul următor se află programarea modulară care permite ascunderea datelor şi a procedurilor în module. Ea este suficientă pentru programarea problemelor relativ simple care nu implică utilizări multiple ale conceptelor modelate prin module. De exemplu, dacă conceptul de stivă este realizat printr-un modul, atunci programarea modulară este suficientă dacă în program nu se utilizează în acelaşi timp mai multe stive de felul celei definite prin modulul respectiv.

Ambele stiluri (programarea procedurală şi cea modulară) sînt suportate de limbajul C.

Nivelul următor, programarea prin abstractizarea datelor, înlătură inconvenientele programării modulare pe baza modelării conceptelor ce intervin în rezolvarea problemelor prin tipuri abstracte de date. De exemplu, introducînd tipul abstract de date "stiva", se pot defini şi utiliza simplu mai multe stive simultan în acelaşi program.

Programarea prin abstractizarea datelor nu este suportată de limbajul C.

Nivelul următor îl constituie programarea orientată spre obiecte. Acest stil de programare, pe lîngă faptul că permite modelarea conceptelor prin tipuri abstracte de date, ea permite în plus şi alte facilităţi dintre care mai sus s-a amintit exprimarea ierarhiilor din cadrul conceptelor prin facilitatea de moştenire.

Programarea prin abstractizarea datelor este suficientă pentru probleme în care intervin concepte individuale care nu au elemente comune şi care deci nu conduc la existenţa unei ierarhii.

Limbajul C++ a fost proiectat pentru a permite definirea şi utilizarea tipurilor abstracte de date, precum şi ierarhizarea lor prin folosirea conceptului de moştenire.

Programarea orientată spre obiecte implică şi alte facilităţi care însă nu sînt suportate direct de limbajul C++. De altfel, la ora actuală nu există un limbaj care să suporte toate conceptele programării orientate spre obiecte.

În cartea de faţă se descriu elementele de bază ale limbajului C++ şi totodată se dau exemple de utilizare a lor în programare.

Cartea apare în trei volume intitulate:

Limbajul C - volumul 1;
Limbajul C++ - volumul 2;
Limbajele C şi C++ în aplicaţii - volumul 3.

În primul volum se face o descriere elementară a limbajului C. Această descriere este însoţită de o serie de exemple şi programe simple care permit o mai bună înţelegere şi fixare a elementelor limbajului C.

Majoritatea exemplelor au un caracter didactic. Cu toate acestea, cititorul va întîlni unele exemple şi programe sau funcţii care pot interveni în practica de zi cu zi a programării în limbajul C.

Autorul insistă asupra importanţei rulării programelor prezentate în carte şi propune ca ele să fie rulate şi înţelese pe măsură ce se avansează cu citirea cărţii.

Programele din primul volum pot fi compilate şi executate folosind mediile integrate de dezvoltare Turbo C sau Turbo C++ implementate de firma BORLAND.

În cazul în care se foloseşte mediul Turbo C se va folosi extensia .C la fişierele sursă, iar în cazul lui Turbo C++ extensia .CPP.

Programele din volumul 2 se pot compila şi executa numai sub mediul Turbo C++ şi fişierele sursă vor avea extensia .CPP.

În volumul 3 se întîlnesc atît programe care nu utilizează elemente specifice limbajului C++ cît şi programe care utilizează astfel de elemente. Programele din prima categorie pot fi compilate şi executate folosind ambele medii amintite mai sus. Evident cele din categoria a doua implică utilizarea mediului Turbo C++.

Volumul 2, conţine o descriere elementară a facilităţilor oferite de limbajul C++, care este însoţită de numeroase exemple şi programe menite să faciliteze înţelegerea şi fixarea cunoştinţelor respective.

Abordarea programelor din volumul 3 este posibilă numai după însuşirea temeinică a cunoştinţelor şi facilităţilor limbajelor C şi C++ prezentate în primele două volume ale cărţii.

Autorul recomandă programatorilor începători să parcurgă volumele 1 - 2 de mai multe ori înainte de a trece la studierea programelor din volumul 3.

Limbajele C şi C++ avînd un caracter universal, volumul 3 conţine programe pentru rezolvări de probleme de sistem, probleme orientate spre calcule ştiinţifice sau prelucrări de date.

Limbajul C se utilizează în aplicaţii care nu implică elemente specifice programării orientate spre obiecte:

- tipuri abstracte de date;
- moştenire;
- supraîncărcarea operatorilor etc.

Majoritatea problemelor de acest fel sînt orientate spre calcule ştiinţifice, ca de exemplu:

- rezolvări de ecuaţii;
- sisteme de ecuaţii liniare;
- probleme de optimizări etc.

În schimb, aplicaţiile de sistem şi de grafică se abordează utilizînd facilităţile programării orientate spre obiecte.

1. NOŢIUNI DE BAZĂ

Un program conţine una sau mai multe *funcţii*. Dintre acestea, una este funcţia *principală*.

Fiecare funcţie are un *nume*. Numele funcţiei principale este *main*. Celelalte funcţii au nume definite de utilizator.

Programul se păstrează într-un *fişier* sau mai multe. Fişierele au extensia *.c* pentru limbajul C şi *.cpp* pentru limbajul C++.

Un fişier care conţine un program scris în C sau C++ sau care conţine numai o parte a acestuia se va numi *fişier sursă*. Prin compilarea unui fişier sursă rezultă un *fişier obiect*. Acesta are extensia *.obj*.

Fişierele sursă care intră în compunerea unui program pot fi compilate împreună sau separat. În urma unei compilări rezultă un fişier obiect. Fişierele obiect, corespunzătoare unui program, pot fi reunite într-un program executabil prin *ediţia de legături* (*link-editare*). În urma link-editării rezultă un *fişier executabil*. Acesta are extensia *.exe*.

1.1. Nume

Un *nume* este o succesiune de litere şi eventual şi cifre, primul caracter fiind literă. În calitate de *litere* se pot utiliza *literele mici* şi *mari* ale alfabetului englez, precum şi caracterul *subliniere* (_).

Numărul de caractere care intră în compunerea unui nume nu este limitat.

În mod implicit, numai primele 32 de caractere dintr-un nume sînt luate în seamă. Aceasta înseamnă că două nume diferă între ele numai dacă ele diferă în primele 32 de caractere ale lor.

Menţionăm că sistemele integrate de dezvoltare Turbo C şi Turbo C++ permit utilizatorului să modifice această limită de 32.

Exemple de nume:

x
i
a1
a_1
a1b2c3
AxY
_ (caracterul subliniere)

_1
a_
acesta_este_un_nume
Acesta_Este_Tot_un_NUME

Se recomandă ca numele să fie sugestiv, adică el să sugereze pe cît posibil scopul alegerii lui sau a datei pe care o reprezintă.

La scrierea numelor se folosesc frecvent literele mici. Adesea, cînd un nume se formează din concatenarea mai multor prescurtări de cuvinte, fiecare cuvînt începe în numele respectiv cu o literă mare.

Exemple:

prodScal sau ProdScal - pentru produs scalar;
prodMat sau ProdMat - pentru produsul a două matrice.

1.2. Cuvinte cheie

Există un număr de cuvinte împrumutate din limba engleză care au o utilizare predefinită. Utilizatorul nu poate să dea o altă utilizare acestor cuvinte. Ele se numesc *cuvinte cheie*.

Cuvintele cheie se scriu cu litere mici. Deci cuvintele cheie sînt nume cu destinaţii speciale.

Exemple:

if
while
for
break
class etc.

Sensul cuvintelor cheie va fi explicat pe măsură ce se vor descrie construcţiile în care ele apar.

1.3. Tipuri de date de bază

Orice limbaj de programare oferă programatorului un număr de tipuri de date de bază.

Tipurile de bază se specifică prin cuvinte cheie. Acestea, în ordine alfabetică sînt:

char
double

float
int
long
short
signed
unsigned

În tabela de mai jos se indică reprezentarea tipurilor de bază.

Reprezentarea tipurilor de bază - Tabela 1

Specificarea tipului	**Dimensiune în biţi**	**Modul de reprezentare**
int	16	întreg reprezentat prin complement faţă de doi
short	16	idem
long	32	idem
unsigned	16	întreg fără semn
unsigned long	32	idem
char	8	codul ASCII al caracterului
float	32	reprezentare flotantă în simplă precizie
double	64	reprezentare flotantă în dublă precizie
long double	80	reprezentare flotantă în dublă precizie

Datele de tip caracter pot fi specificate prin:

unsigned char

sau

signed char

În primul caz, data se presupune că este un întreg în intervalul [0,255] (întreg fără semn).

În cel de al doilea caz, data se presupune că aparţine intervalului [-128,127] (întreg cu semn).

Datele specificate numai prin *char* au o interpretare implicită. Această interpretare poate fi definită prin intermediul mediului integrat de dezvoltare Turbo C sau Turbo C++. În mod normal se consideră că datele de tip *char* sînt implicit numere fără semn.

Pentru a preîntîmpina apariţia unor erori prin interpretări implicite nedorite ale datelor de tip caracter, se recomandă a se utiliza specificările explicite *unsigned char* şi *signed char*.

În tabela 2 se indică intervalele de valori ale tipurilor de bază.

Intervalele de valori pentru tipurile de bază - Tabela 2

Specificarea tipului	**Intervalul de valori**
int	[-32768,32767]
short	[-32768,32767]
long	[-2147483648,2147483647]
unsigned	[0,65535]
unsigned long	[0,4294967295]
unsigned char	[0,255]
signed char	[-128,127]
float	valoarea absolută a unei date diferite de zero aparţine intervalului [3,4*10**(-38);3,4*10**(38)]
double	valoarea absolută a unei date diferite de zero aparţine intervalului [1,7*10**(-308);1,7*10**(308)]
long double	valoarea absolută a unei date diferite de zero

continuare

Specificarea tipului	Intervalul de valori
	aparţine intervalului [3,4*10**(-4932);1,1*10**(4932)]

Observaţie:

Tipul predefinit *short* este identic cu tipul predefinit int la calculatoarele compatibile IBM PC. La alte calculatoare ele pot fi diferite, de exemplu tipul *short* rămîne reprezentat pe 16 biţi, iar tipul int poate să fie la fel ca şi tipul *long*.

Tipul *unsigned long* poate fi scris inversînd cuvintele cheie:

long unsigned.

1.4. Structura unei funcţii

O funcţie are următoarea structură:

tip nume(lista declaraţiilor parametrilor formali)
{
declaraţii
instrucţiuni
}

Primul rînd din formatul de mai sus reprezintă *antetul* funcţiei. Partea inclusă între acolade, împreună cu acoladele, formează *corpul* funcţiei.

În cazul tipurilor predefinite, *tip* din antetul funcţiei este un cuvînt cheie. El defineşte tipul valorii returnate de funcţie.

În limbajul C există două categorii de funcţii. O primă categorie conţine funcţiile care la revenirea din ele returnează o valoare în punctul de apel. Tipul acestei valori se defineşte prin tip din antetul funcţiei. Cealaltă categorie de funcţii conţine funcţiile care nu returnează nici o valoare la revenirea din ele.

Pentru aceste funcţii se va folosi cuvîntul cheie *void* în calitate de *tip*. El semnifică lipsa unei valori returnate la revenirea din funcţie.

O funcţie poate avea zero sau mai mulţi parametri. Lista declaraţiilor parametrilor formali este vidă în cazul în care funcţia nu are parametri formali. În acest caz antetul funcţiei se reduce la:

tip nume()

Menţionăm că absenţa parametrilor poate fi indicată explicit folosind cuvîntul cheie *void*. Astfel, antetul de mai sus poate fi scris şi sub forma:

tip nume(void)

Exemple:

1. **void** *f*(**void**)

```
{
...
}
```

Funcţia *f* nu are parametri. Ea nu returnează nici o valoare la revenirea din ea.

2. **void** *f*()

```
{
...
}
```

Format identic cu cel din exemplul precedent.

3. **int** *g*()

```
{
...
}
```

Funcţia *g* nu are parametri. La revenirea din ea se returnează o valoare întreagă de tip *int*.

4. **double** *h*(**void**)

```
{
...
}
```

Funcţia *h* nu are parametri. La revenirea din ea se returnează o valoare flotantă în dublă precizie.

În cazul în care funcţia are parametri, declaraţiile parametrilor formali se includ între parantezele rotunde prezente după numele funcţiei şi se separă prin virgulă dacă sînt mai multe.

Parametri se utilizează pentru a permite transferuri de date la o funcţie în momentul apelului ei. Acest mecanism de transfer al datelor prin

intermediul parametrilor ne permite construirea de funcţii făcînd abstracţie de valorile concrete care vor fi prezente abia la execuţia programului. În momentul compilării este necesară numai cunoaşterea *tipurilor* valorilor pe care le vor primi parametri la execuţie. Aceste tipuri sînt definite prin declaraţiile parametrilor respectivi, declaraţii care, aşa cum am văzut mai sus, se indică în antetul funcţiei.

Parametri declaraţi în antetul unei funcţii şi care apoi se utilizează în corpul funcţiei, se numesc *formali* pentru a sublinia faptul că ei nu reprezintă valori concrete, ci numai ţin locul acestora pentru a putea exprima procesul de calcul realizat prin funcţie. Ei se concretizează la execuţie prin apelurile funcţiei. Valorile parametrilor formali se definesc la fiecare apel al unei funcţii prin aşa numiţii parametri *reali*, *efectivi* sau *concreţi*.

Utilizarea parametrilor formali la implementarea funcţiilor şi atribuirea de valori concrete pentru ei la execuţie, reprezintă un prim nivel de abstractizare în programare. Acest mod de programare este cel mai vechi. El se numeşte *programare procedurală* şi realizează un proces de *abstractizare prin parametri*.

Declaraţiile parametrilor formali sînt asemănătoare cu cele ale variabilelor şi vor fi definite într-un paragraf ulterior.

Observaţii:

1. Iniţial antetul unei funcţii a avut următorul format:

   tip nume(lista parametrilor formali)

   declaraţiile parametrilor formali

   unde *tip* este prezent numai pentru funcţii care returnează o valoare la revenirea din ele. Cuvîntul cheie *void* a fost introdus ulterior. De asemenea, cuvîntul cheie *int* nu era nevoie să fie prezent, considerîndu-se că orice funcţie care returnează o valoare a cărui tip nu este specificat, returnează o valoare de tip *int*.

Ulterior s-a constatat că această libertate în omiterea tipului este o sursă de erori şi de aceea se recomandă ca tipul să fie totdeauna prezent în antetul unei funcţii. În cazul limbajului C++ controalele cu privire la tipul valorii au fost întărite chiar pentru a elimina posibilităţile de apariţie a erorilor la revenirea din funcţii.

Lista parametrilor formali este fie vidă, cînd funcţia nu are parametri, fie se compune dintr-un nume sau mai multe separate prin virgulă. În acest caz, declaraţiile parametrilor formali, dacă ei există, se dau imediat după

paranteza închisă. Acest format poate fi folosit, atît în limbajul C, cît şi în limbajul C++, dar compilatorul C++ remarcă printr-un avertisment că acest format este învechit.

De asemenea, amintim că cele două formate pot fi folosite împreună, adică putem indica în parantezele rotunde o parte din parametri prin numele lor, iar restul prin declaraţiile lor.

În continuare se dau declaraţiile parametrilor care în parantezele rotunde au fost prezente numai prin numele lor.

Cu toate acestea, se recomandă utilizarea formatului în care toate declaraţiile parametrilor sînt incluse în parantezele rotunde (formatul de la începutul paragrafului).

2. Pentru funcţia principală se pot utiliza antetele:

```
int main()
int main(void)
void main()
void main(void)
    main()
    main(void)
```

Primele două antete presupun că funcţia *main* returnează o valoare întreagă la revenirea din ea în sistemul de operare.

Adesea se obişnuieşte utilizarea formatului fără tip:

main()

Amintim că funcţia *main* poate avea şi parametri. Aceştia, cînd sînt prezenţi, permit utilizarea de către program a unor valori definite la lansarea programului. Ulterior vom preciza modul de utilizare al acestor parametri.

1.5. Comentariu

În limbajul C, ca şi în alte limbaje de programare se pot folosi comentarii.

Un comentariu începe cu succesiunea de caractere

/*

şi se termină cu

*/

El se compune din orice caracter admis în setul de caractere al limbajului. Evident, în interiorul unui comentariu nu se va folosi succesiunea

*/

care termină un comentariu.

În limbajul C++ s-a mai introdus o convenţie pentru a insera comentarii. Astfel, în C++ un comentariu poate începe prin succesiunea

//

Un astfel de comentariu se termină pe acelaşi rînd (la sfîrşitul lui) pe care se află şi începutul lui.

Comentariile sînt explicaţii pentru programatori. Ele nu au nici un efect asupra compilatorului şi sînt omise la compilare.

Un comentariu se poate insera oriunde în program unde este legal să apară un spaţiu, un tabulator sau caracterul de rînd nou.

În general, se recomandă introducerea de comentarii după antetul funcţiei, care să precizeze:

- acţiunea sau acţiunile realizate de funcţie;
- formatele datelor de intrare şi ieşire;
- algoritmii codificaţi prin funcţia respectivă dacă sînt complecşi;
- diferite limite impuse datelor de intrare etc.

De asemenea, se pot insera comentarii în corpul funcţiei unde se consideră că sînt necesare unele explicaţii.

Comentariile trebuie să fie exprimări clare care să nu conducă la amibiguităţi şi să nu conţină afirmaţii eronate.

1.6. Constante

O constantă are un *tip* şi o *valoare*. Atît tipul, cît şi valoarea, sînt determinate de caracterele care intră în compunerea constantei. Valoarea unei constante nu poate fi schimbată în timpul execuţiei programului în care a fost utilizată.

1.6.1. Constante întregi

Constantele întregi pot fi scrise în sistemul de numeraţie cu baza 8, 10

sau 16.

O *constantă zecimală întreagă* este un şir de cifre zecimale care are prima cifră diferită de zero. Constantele zecimale se reprezintă prin complement faţă de doi pe 16 biţi sau pe 32 de biţi dacă nu încap pe 16 biţi.

Constantele întregi reprezentate pe 16 biţi sînt de tip *int*, iar cele reprezentate pe 32 biţi sînt de tip *long*.

În cazul în care noi dorim să reprezentăm o constantă zecimală pe 32 de biţi, chiar dacă ea se poate reprezenta pe 16 biţi, constanta respectivă trebuie să o terminăm prin *L* sau *l*.

Exemple:

Reprezentare externă	**Reprezentare internă în binar**
12345	0011000000111001
123456789	00000111010110111100110100010101
12345L	00000000000000000011000000111001

O constantă zecimală are tipul *unsigned* dacă se termină prin litera *U* sau *u*. O astfel de constantă se reprezintă pe 16 biţi dacă nu este mai mare decît 65535 şi pe 32 de biţi în caz contrar. În cazul în care dorim ca o constantă întreagă fără semn mai mică decît 65536 să se reprezinte pe 32 de biţi, constanta respectivă se va termina prin una din următoarele succesiuni de litere:

ul
lu
LU
UL

Constantele întregi fără semn pot fi utilizate pentru a economisi memorie. Astfel, constantele de tip *int* din intervalul [32768,65535] se păstrează pe 32 de biţi, în schimb constantele de tip *unsigned* din acelaşi interval se reprezintă pe 16 biţi.

Exemple:

40000u	constantă întreagă de tip *unsigned* reprezentată pe 16 biţi
40000U	idem

4294967295	constantă întreagă de tip *unsigned* reprezentată pe 32 de biţi
40000lu	idem
40000ul	idem
40000LU	idem
40000UL	idem

O *constantă octală* întreagă este o succesiune de cifre octale (0 - 7) precedată de un zero nesemnificativ. O astfel de constantă se păstrează pe 16 biţi dacă aceştia îi sînt suficienţi şi pe 32 de biţi în caz contrar. În cazul în care o constantă octală se termină prin *l* sau *L*, ea se păstrează pe 32 de biţi chiar dacă sînt suficienţi 16 biţi pentru reprezentarea ei.

Constantele octale sînt de tip *unsigned* dacă se reprezintă pe 16 biţi şi *unsigned long* dacă se reprezintă pe 32 de biţi.

O *constantă hexazecimală* întreagă este o succesiune de cifre hexazecimale precedată de

0x

sau

0X.

În rest, ea are aceleaşi proprietăţi ca şi o constantă octală.

Cifrele hexazecimale se obţin extinzînd cifrele zecimale cu literele mici sau mari de la *A* la *F*:

Litera care reprezintă o cifră hexazecimală	**Valoare**
a sau A	10
b sau B	11
c sau C	12
d sau D	13
e sau E	14
f sau F	15

Exemple:

Constanta	Tipul constantei	Lungimea reprezentării
123	constantă zecimală de tip int	16 biţi
0123	constantă octală de tip unsigned	16 biţi
40000	constantă zecimală de tip long	32 biţi
040000	constantă octală de tip unsigned	16 biţi
0123456	idem	idem
123L	constantă zecimală de tip long	32 biţi
0123l	constantă octală de tip unsigned long	32 biţi
0x123	constantă hexazecimală de tip unsigned	16 biţi
0xa1b2c3	constantă hexazecimală de tip long unsigned	32 biţi
0XABCFL	constantă hexazecimală de tip long unsigned	32 biţi

1.6.2. Constante flotante

O *constantă flotantă* reprezintă un număr *raţional*. Ea se compune din:

- o parte întreagă care poate fi şi vidă;
- o parte fracţionară care poate fi şi vidă;
- un exponent care poate fi şi vid.

Evident, nu pot fi vide toate părţile indicate mai sus. La scrierea unei constante flotante este necesar să fie prezentă fie partea fracţionară, fie exponentul precedat de partea întreagă.

Partea întreagă este o constantă zecimală.

Partea fracţionară se compune din caracterul punct după care urmează o succesiune de cifre zecimale. Succesiunea respectivă poate fi vidă numai în cazul în care partea întreagă este prezentă.

Exponentul începe cu litera e mică sau mare, după care poate fi prezent un semn opţional (plus sau minus) şi un şir de cifre zecimale. Exponentul defineşte un factor care exprimă o putere a lui 10.

În exemplele de mai jos şi în continuare vom folosi notaţia ** pentru operaţia de ridicare la putere. Deci *a* la puterea *b* se va nota prin

a**b

Exemple:

Constantă flotantă	**Valoare**
123.	123
123.7	123,7
.25	0,25
78e4	78*10**4
.1E-3	0,1*10**(-3)
123.456e2	12345,6
1234.567e-4	0,1234567

Constantele flotante se reprezintă în dublă precizie (64 biţi).

Pentru a reprezenta o constantă flotantă în simplă precizie este suficient să terminăm constanta respectivă prin litera *f* sau *F*.

O constantă flotantă terminată prin *l* sau *L* se reprezintă pe 80 de biţi şi are tipul *long double*.

1.6.3. Constantă caracter

Prelucrarea datelor cu ajutorul calculatoarelor are în vedere, printre altele, posibilitatea lucrului pe caractere. În acest scop, caracterele se codifică folosind coduri numerice. Cele mai utilizate coduri sînt codurile EBCDIC (Extended Binary Coded Decimal Interchange Code) şi ASCII (American Standard Code for Information Interchange).

La calculatoarele compatibile IBM PC se utilizează codul ASCII. Caracterele acestui cod le împărţim în:

caractere negrafice - Codul lor este în intervalul [0,31] la care se adaugă şi codul 127 (caracterul DEL).

spaţiu - Are codul 32.

caractere grafice - Codul lor este în intervalul [33,126].

Caracterele grafice împreună cu spaţiul formează setul de caractere *imprimabile*.

O constantă caracter are ca valoare codul ASCII al caracterului pe care-l reprezintă. Ea are tipul *int*.

O constantă caracter corespunzătoare unui caracter imprimabil se reprezintă prin caracterul respectiv inclus între caractere apostrof.

Exemple:

Constantă caracter	Valoare
'a'	97
'A'	65
'0'	48
' '	32
'*'	42

O excepţie de la regula de mai sus o reprezintă caracterele apostrof şi bara oblică inversă (*backslash*). Astfel, caracterul *backslash* se reprezintă prin două caractere backslash incluse între caractere apostrof:

'\\'

La reprezentarea caracterului apostrof se utilizează caracterul backslash urmat de un caracter apostrof:

\'

De aceea, constanta caracter apostrof se poate reprezenta prin:

'\''

Caracterul backslash se poate utiliza pentru a defini constante caracter şi pentru caractere negrafice. Aşa de exemplu, pentru a defini constanta caracter corespunzătoare tabulatorului se foloseşte notaţia cu backslash:

\t

Deci

'\t'

defineşte constanta caracter tabulator orizontal. Ea are valoarea 9.

În mod analog, constanta caracter rînd nou (*newline*) se reprezintă prin:

'\n'

Ea are valoarea 10.

Se obişnuieşte să se spună că backslash introduce o secvenţă *escape*.

Mai jos se indică reprezentarea unor constante caracter prin secvenţe *escape*.

Constantă caracter	Codul ASCII	Denumirea caracterului	Utilizare
'\a'	7	BEL	activare sunet
'\b'	8	BS	revenire cu un spaţiu (Backspace)
'\t'	9	HT	tabulator orizontal
'\n'	10	LF	rînd nou (Line Feed)
'\v'	11	VT	tabulator vertical
'\f'	12	FF	salt de pagină la imprimantă (Form Feed)
'\r'	13	CR	retur de car - poziţionează cursorul în coloana 1 din rîndul curent (Carriage Return)

La aceste notaţii adăugăm şi utilizările secvenţei *escape* pentru a reprezenta caracterele backslash, apostrof şi ghilimele:

Constantă caracter	Codul ASCII	Denumirea caracterului
'\"'	34	ghilimele
'\''	39	apostrof
'\\'	92	backslash

Secvenţa *escape* poate fi folosită pentru a defini constante caracter pentru orice caracter al codului ASCII. Aceasta se realizează folosind codul caracterului respectiv. Astfel, construcţia:

'\ddd'

unde:

d - Este o cifră octală.

- Reprezintă constanta caracter corespunzătoare codului ASCII egal cu valoarea întregului octal ddd.

Exemple:

'\a' şi '\7' - Reprezintă aceeaşi constantă caracter corespunzătoare caracterului BEL.

'\b' şi '\10' - Reprezintă constanta caracter corespunzătoare caracterului backspace (8 din sistemul zecimal se reprezintă prin numărul 10 în sistemul cu baza opt).

Se observă că în cadrul unei secvenţe escape numerele octale nu mai trebuie să fie precedate de un zero nesemnificativ. În acest caz numărul se consideră automat în sistemul cu baza 8.

'\"' şi '\42' - Reprezintă constanta caracter care corespunde caracterului ghilimele.

La calculatoarele compatibile IBM PC se utilizeză un cod *ASCII extins*. Acesta conţine 256 de coduri care sînt valori în intervalul [0,255].

Codurile din intervalul [0,127] corespund caracterelor codului ASCII obişnuit, iar cele din intervalul [128,255] corespund unor caractere care au o utilizare specială.

Constantele caracter corespunzătoare caracterelor de cod ASCII din intervalul [128,255] se definesc prin secvenţe escape în care se indică codul acestora. De exemplu, constanta caracter

'\377'

corespunde caracterului care are codul ASCII egal cu 255, adică codul ASCII maxim.

Deoarece o constantă caracter este de tip *int*, ea se păstrează pe 16 biţi. Pentru constantele caracter mai mari decît 127, se pune problema extensiei semnului. Programatorul poate opta pentru constante caracter cu semn sau fără semn. Pentru a folosi în mod implicit constante caracter fără semn se va alege, în mediul integrat de dezvoltare Turbo C sau Turbo C++, alternativa *Unsigned Characters* în submeniul *Code Generation* al submeniului *Compile* din meniul *Options*.

1.6.4. Şir de caractere

O succesiune de zero sau mai multe caractere incluse între ghilimele formează o constantă şir sau un şir de caractere.

La scrierea caracterelor din compunerea unui şir de caractere se pot utiliza secvenţe escape.

Exemple:

```
"Acesta este un sir de caractere"
"Prin secventa escape \" se reprezinta ghilimelele"
"Prin secventa escape \\ se reprezinta backslash"
"Apostroful se reprezinta obisnuit"
"s'a"
"alte\tsecvente\tescape\nintr-un sir"
```

Un şir poate fi continuat pe rîndul următor folosind caracterul backslash. În acest scop se tastează backslash la sfîrşitul rîndului care se continuă, se trece pe rîndul următor (acţionînd tasta Enter) şi se continuă cu tastarea caracterelor şirului respectiv.

Caracterul care precede pe backslash se va concatena cu primul caracter de pe rîndul următor.

Caracterele unui şir de caractere se păstrează în memorie într-o zonă contiguă, prin codurile lor ASCII. După ultimul caracter al şirului se păstrează caracterul *NUL*, adică valoarea zero. Acesta joacă rolul de marcaj de sfîrşit al oricărui şir de caractere. Din cauza acestui marcaj, trebuie să facem distincţie între o constantă caracter care corespunde unui caracter şi şirul de caractere care este format din acelaşi caracter. Astfel, constanta caracter

'A'

se păstrează în memorie într-un octet prin valoarea 65, pe cînd şirul de caractere

"A"

ocupă o zonă de doi octeţi: în primul octet se păstrează valoarea 65, iar în al doilea caracterul *NUL*, adică valoarea zero.

În concluzie, un şir de caractere se păstrează în memorie într-o succesiune de octeţi al cărui număr este egal cu numărul caracterelor şirului respectiv mărit cu 1, deoarece şirul se termină totdeauna prin caracterul NUL.

Observaţii:

1. Fie şirul

   "a\1b"

   Acest şir are în compunerea sa:

   - caracterul a de cod ASCII 97;
   - caracterul SOH de cod ASCII 1;
   - caracterul b de cod ASCII 98;
   - caracterul NUL de cod ASCII 0.

Dacă în locul şirului de mai sus se doreşte un şir în care *b* să fie înlocuit prin caracterul 3, atunci scrierea:

"a\13"

nu este corectă. Într-adevăr, acest şir are în compunerea sa:

- caracterul a de cod ASCII 97;
- caracterul VT de cod ASCII 11;
- caracterul NUL de cod ASCII 0.

Pentru a reprezenta caracterul 3 în acest caz, va fi nevoie de încă o secvenţă escape. Secvenţa escape pentru caracterul 3 este \63, deci şirul de caractere respectiv se scrie astfel:

"a\1\63"

2. Şirul de caractere

   "\1751"

   are în compunerea sa:

   - caracterul } de cod ASCII 125 (175 în octal);
   - caracterul 1 de cod ASCII 49;
   - caracterul NUL de cod ASCII 0.

   El poate fi scris mai simplu astfel:

   "}1"

3. Caracterul NUL nu poate fi utilizat decît la sfîrşitul unui şir de caractere. Aceasta, deoarece un şir de caractere totdeauna se termină la apariţia caracterului NUL.

1.7. Caractere sau spaţii albe (white spaces)

În cele ce urmează, prin caracter sau spaţiu alb vom înţelege unul din următoarele caractere:

- spaţiu (' ');
- tabulator orizontal ('\t');
- caracterul de rînd nou (newline) ('\n').

Menţionăm că setul caracterelor albe diferă pentru diferite implementări ale limbajului C. Setul de caractere indicat mai sus este un set minimal comun tuturor implementărilor.

Amintim că un comentariu poate fi inserat într-un program, oriunde este legal să apară un spaţiu alb.

1.8. Variabile simple, tablouri şi structuri

Într-un program utilizăm alături de date constante şi date variabile care îşi schimbă valorile în timpul execuţiei programului.

Dacă la o dată constantă ne putem referi folosind caracterele din compunerea ei, la o dată variabilă trebuie să ne referim altfel. Cel mai simplu mod este acela de a denumi data respectivă. Numele datei ne permite accesul la valoarea ei, precum şi schimbarea valorii dacă este necesar.

În cazul în care o dată nu are legături cu alte date (de exemplu de ordine), vom spune că ea este o dată *izolată*. Numele unei date izolate se spune că reprezintă o *variabilă simplă*.

Unei date izolate îi corespunde un *tip*. În cursul execuţiei programului se pot schimba valorile unei date variabile dar nu şi tipul ei.

Corespondenţa dintre numele unei date variabile şi tipul ei se defineşte printr-o *declaraţie*.

Adesea, într-un program este util să considerăm grupe de date. Gruparea datelor se poate face în mai multe moduri.

Un mod simplu de a grupa date este acela de a considera date de *acelaşi* tip, în aşa fel încît grupa respectivă să formeze o *mulţime ordonată* de elemente la care să ne putem referi folosind *indici*. O astfel de grupă se spune că formează un *tablou*. Unui tablou i se dă un nume. Tipul comun al elementelor unui tablou este şi tipul tabloului respectiv. De exemplu, o mulţime ordonată de întregi reprezintă un tablou de tip întreg.

În cazul în care elementele care se grupează într-un tablou sînt ele însele tablouri, vom avea nevoie de mai mulţi indici, pentru a ne referi la ele. În cazul în care se utilizează un singur indice pentru a ne referi la elementele tabloului, spunem că tabloul este *unidimensional*. Dacă se folosesc n indici, se spune că tabloul este *n-dimensional*.

Exemple simple de tablouri unidimensionale sînt vectorii care au componente de acelaşi tip. Aşa de exemplu, un vector de componente întregi este un tablou unidimensional de tip întreg.

O matrice de elemente întregi este un exemplu de tablou bidimensional de tip întreg.

Referirea la elementele unui tablou se face printr-o *variabilă cu indici*.

O variabilă cu indici se compune din numele tabloului urmat de valorile indicilor, fiecare indice fiind reprezentat printr-o expresie inclusă între paranteze pătrate.

Valoarea *inferioară* a indicilor este egală cu *zero*. De exemplu, dacă *vect* este un tablou unidimensional de 10 elemente, atunci ne referim la elementele lui cu ajutorul variabilelor cu indici:

vect[0]	primul element
vect[1]	al doilea element
...	
vect[9]	ultimul element

Dacă *mat* este un tablou bidimensional care defineşte o matrice de 3 linii a 2 coloane fiecare, atunci elementele acestui tablou pot fi referite prin:

mat[0][0] mat[0][1]	elementele primei linii
mat[1][0] mat[1][1]	elementele celei de a doua linii
mat[2][0] mat[2][1]	elementele celei de a treia linii

Un alt mod de a grupa date are în vedere prezenţa unor *relaţii* între datele care se grupează. În acest caz datele care se grupează se spune că formează o *structură*. Ele nu neapărat au un acelaşi tip.

Prin *structură* înţelegem o mulţime ordonată de elemente "înrudite", ordonare definită de utilizator în vederea simplificării utilizării grupei de date respective.

Unei structuri i se ataşează un *nume*. De asemenea, se ataşează cîte un nume fiecărei componente ale unei structuri, nume care se utilizează la referirea componentelor respective.

Un exemplu simplu de structură este data calendaristică. Aceasta este o grupă care se compune din trei date înrudite:

- zi;
- lună;
- an.

Aceste trei date nu neapărat au toate trei acelaşi tip. De exemplu, ziua şi anul pot fi de tip *întreg* (*int*), iar luna poate fi de tip *nenumeric* dacă ea se reprezintă prin denumire.

Un alt exemplu simplu de structură îl reprezintă numerele complexe. Un număr complex este o mulţime ordonată de două numere, fiecare de tip *double*. Primul număr reprezintă partea reală a numărului complex, iar cel de al doilea, partea lui imaginară.

Structura, ca şi variabilele simple şi tablourile, corespunde şi ea unui tip de dată. Tipul unei structuri nu are nimic comun cu tipurile componentelor sale. Ea este un tip *nou*, diferit de cele predefinite în limbaj. Un astfel de tip se spune că este un *tip definit de utilizator* sau mai scurt *tip utilizator*. De

exemplu, data calendaristică definită mai sus reprezintă un tip nou de date, diferit de cele predefinite. De asemenea, numerele complexe reprezentate prin structuri de perechi de numere flotante în dublă precizie, definesc tipul complex.

Tipurile utilizator se definesc printr-o *declaraţie* care, aşa cum vom vedea ulterior, a suferit mai multe modificări. Tot printr-o declaraţie se stabileşte legătura dintre numele unei date structurate şi tipul ei.

În general, prin *tip* înţelegem o mulţime de date împreună cu operaţiile care pot fi efectuate cu datele respective.

De exemplu, tipul *int* se defineşte prin mulţimea numerelor întregi din intervalul [-32768,32767] reprezentate prin complement faţă de doi pe 16 biţi. Asupra acestor date sînt definite o serie de operaţii, cum ar fi cele 4 operaţii aritmetice, operaţii de comparaţie etc.

În cazul tipurilor predefinite sînt predefinite atît mulţimea şi reprezentarea datelor tipului respectiv, cît şi operaţiile cu aceste date.

În cazul tipurilor utilizator se definesc de către utilizator, mulţimea şi reprezentarea datelor tipului respectiv prin intermediul construcţiei *struct*. Utilizatorul defineşte operaţiile cu aceste date prin intermediul unor funcţii. De exemplu, în cazul tipului *complex* se defineşte reprezentarea numerelor complexe printr-o construcţie *struct*. Operaţiile cu numerele complexe se pot realiza apelînd funcţii corespunzătoare.

Tipurile utilizator pot fi definite ca mai sus prin intermediul facilităţilor existente în limbajul C. Un neajuns al tipurilor utilizator definite în limbajul C este faptul că nu se stabileşte nici o legătură între reprezentarea tipului şi funcţiile care definesc operaţiile cu datele respective. Acest neajuns a fost înlăturat în C++ prin introducerea noţiunii de *clasă*. Tipurile definite prin intermediul claselor se numesc *tipuri abstracte* de date.

În cazul unui tip abstract de date se asigură legătura dintre reprezentările datelor şi funcţiile care definesc operaţiile asupra lor. Ca rezultat al acestei legături se obţine protecţia datelor, utilizatorul neavînd acces direct la ele decît numai prin intermediul funcţiilor definite în acest scop.

Construcţia *struct*, precum şi definirea claselor se vor prezenta mai tîrziu.

1.9. Declaraţia de variabilă simplă

În limbajul C nu există declaraţii implicite. Aceasta înseamnă că orice variabilă înainte de a fi utilizată trebuie declarată.

Declaraţia unei variabile simple stabileşte legătura dintre numele variabilei şi tipul valorilor pe care le poate avea variabila respectivă.

În cea mai simplă formă, o declaraţie de variabilă simplă are formatul:

tip lista_de_nume;

În calitate de *tip* putem folosi cuvintele cheie ale tipurilor predefinite.

Lista_de_nume se compune dintr-un nume de variabilă simplă sau mai multe separate prin virgule.

Exemple:

1. int i,j;

   Variabilele *i* şi *j* sînt variabile simple de tip *int*. Compilatorul alocă pentru fiecare o zonă de memorie de 16 biţi.

   Prin *i* ne referim la valoarea conţinută în zona de memorie alocată variabilei *i*. De asemenea, tot prin intermediul lui *i* putem atribui sau modifica valoarea din zona de memorie alocată variabilei *i*.

2. char c;

   Variabila *c* este o variabilă simplă de tip *char*. Ei i se alocă o zonă de memorie de un octet (8 biţi).

   În această zonă putem păstra un caracter al codului ASCII extins, prin codul lui. Prin *c* ne putem referi la caracterul respectiv.

3. long double x;

   Variabila *x* este o variabilă simplă de tip *long double*. Ei i se alocă o zonă de memorie de 10 octeţi în care se păstrează o valoare flotantă.

1.10. Declaraţia de tablou

Un tablou, ca orice variabilă simplă, trebuie declarat înainte de a fi utilizat. Declaraţia de tablou, în forma cea mai simplă, conţine *tipul comun* elementelor sale, *numele* tabloului şi *limitele superioare* pentru fiecare indice, incluse între paranteze pătrate:

tip nume[lim1][lim2]...[limn];

unde:

tip - Este un cuvînt cheie pentru tipurile predefinite.

limi - Este limita superioară a indicelui al i-lea; aceasta înseamnă că indicele al i-lea poate avea valorile:

0,1,2,...,limi-1

Limitele *limi* (i=1,2,...,n) sînt *expresii constante*. Prin expresie constantă înţelegem o expresie care poate fi evaluată la compilare în momentul întîlnirii ei de către compilator.

În paragraful precedent am văzut că la elementele unui tablou ne putem referi folosind variabile cu indici.

În limbajul C, numele unui tablou este un simbol care are ca valoare adresa primului său element.

La întîlnirea unei declaraţii de tablou, compilatorul alocă o zonă de memorie necesară pentru a păstra valorile elementelor sale. Numele tabloului respectiv poate fi utilizat în diferite expresii şi valoarea lui este chiar adresa de început a zonei de memorie care i-a fost alocată.

Exemple:

1. int vect[10];

Declaraţia de faţă defineşte tabloul *vect* de 10 elemente şi el are tipul *int*. Pentru acest tablou se alocă 10*2=20 octeţi.

vect este un simbol a cărui valoare este adresa primului său element, adică adresa lui vect[0]. Deci vect[0] are ca valoare valoarea primului element al tabloului, iar *vect* are ca valoare adresa acestui element.

2. char tab[100];

Tabloul *tab* este un tablou unidimensional de tip *char*, care are 100 de elemente. I se alocă 100 de octeţi şi *tab* are ca valoare adresa elementului tab[0].

3. double dmat[10][50];

Tabloul *dmat* este un tablou bidimensional de tip *double*. El reprezintă o matrice de 10 linii a 50 de coloane fiecare. Compilatorul rezervă pentru acest tablou

10*50*8=4000 octeţi.

La elementele acestui tablou ne referim prin:

dmat[0][0] dmat[0][1]...dmat[0][49]

dmat[1][0] dmat[1][1]...dmat[1][49]

...

dmat[9][0] dmat[9][1]...dmat[9][49]

dmat are ca valoare adresa elementului dmat[0][0].

1.11. Apelul şi prototipul funcţiilor

Într-un program o funcţie poate avea o definiţie şi unul sau mai multe apeluri.

Am văzut mai sus că o funcţie se defineşte prin antet urmat de corpul ei. Antetul, de obicei, are formatul:

tip nume(lista declaraţiilor parametrilor formali)

Lista declaraţiilor parametrilor formali este fie vidă, fie conţine o declaraţie de parametru formal sau mai multe separate prin virgulă. Menţionăm că declaraţiile parametrilor formali aflate într-o listă de felul celei de sus, nu se termină prin punct şi virgulă ca cele pentru variabile simple şi tablouri.

Exemple:

1. int f(int x,double y)

   Funcţia *f* are doi parametri:

   x de tip *int*

   şi

   y de tip *double*.

   Funcţia returnează o valoare de tip *int*.

2. double df(long a, int b, unsigned c)

   Funcţia *df* are trei parametri:

   a - de tip *long*;

   b - de tip *int*;

   c - de tip *unsigned*.

   Ea returnează o valoare de tip double.

O funcţie poate fi apelată folosind o construcţie de forma:

nume(lista_parametrilor_efectivi)

unde:

nume	- Este numele funcţiei care se apelează.
lista parametrilor efectivi	- Este fie vidă dacă funcţia nu are parametri, fie se compune din unul sau mai mulţi *parametri efectivi* separaţi prin virgule.
	- Un parametru efectiv este o expresie.
	- Parametri efectivi se corespund cu cei formali prin *ordine* şi *tip*.

La apel se atribuie parametrilor formali valorile parametrilor efectivi şi apoi execuţia se continuă cu prima instrucţiune din corpul funcţiei apelate. La revenirea din funcţie se ajunge în funcţia din care s-a făcut apelul şi execuţia continuă cu construcţia următoare apelului.

Pentru a apela o funcţie putem utiliza construcţia de mai sus urmată de caracterul punct şi virgulă.

O altă posibilitate este aceea de a folosi construcţia de mai sus drept operand al unei expresii. Un astfel de apel este posibil numai pentru funcţiile care returnează o valoare la revenirea din ele. În acest caz valoarea returnată de funcţie se foloseşte la evaluarea expresiei din care s-a făcut apelul.

Un parametru efectiv de la apelul unei funcţii poate fi numele unui tablou. În acest caz, în antetul funcţiei respective parametrul corespunzător îl vom declara ca fiind tablou.

De exemplu, dacă *tab* este un tablou unidimensional declarat ca mai jos:

```
int tab[100];
```

şi *tab* se foloseşte la apelul funcţiei *f*:

```
f(tab);
```

atunci funcţia *f* are antetul:

```
void f(int x[100])
```

Menţionăm că, limita superioară 100 poate fi omisă la declararea parametrului formal *x*, dar nu şi parantezele pătrate. Deci aceeaşi funcţie poate avea următorul antet:

```
void f(int x[])
```

În cazul parametrilor formali care sînt tablouri cu mai multe dimensiuni, numai limita primului indice poate fi omisă.

Exemplu:

Fie declaraţia

double mat[4][10];

şi apelul

fct(mat);

Antetul funcţiei *fct* poate fi următorul:

void fct(double mat[][10])

O funcţie poate fi apelată într-un punct al unui fişier sursă dacă în prealabil a fost definită în acelaşi fişier sursă.

Exemplu:

```
void f1(void) // definitia functiei f1
{
...
}

void f2(void) // definitia functiei f2
{
...
/* se apeleaza functia f1; ea este in prealabil definita */
f1();
...
}
```

Apelurile funcţiei nu pot fi precedate totdeauna de definiţia ei. În astfel de cazuri definiţia funcţiei apelate este înlocuită printr-un aşa numit *prototip* al ei.

Prototipul unei funcţii are un format asemănător cu antetul ei. Acesta reprezintă o informaţie pentru compilator cu privire la:

- tipul valorii returnate de funcţie;
- existenţa şi tipurile parametrilor funcţiei.

Aceste informaţii sînt prezente în antetele funcţiilor. De aceea, un prototip al unei funcţii poate fi scris ca şi antetul funcţiei respective, după care se pune punct şi virgulă. Există şi o formă prescurtată pentru prototip şi anume aceea în care se omite numele parametrilor:

tip nume(lista tipurilor parametrilor formali);

Exemple:

1. void f(void);

Prototipul de faţă indică faptul că *f* este o funcţie fără parametri şi care nu returnează nici o valoare.

2. double a(void);

Funcţia *a* nu are parametri. Ea returnează o valoare flotantă în dublă precizie.

3. void c(int x,long y[],double z);

Funcţia *c* nu returnează nici o valoare. Are trei parametri:

- primul este de tip *int*;
- al doilea este un tablou unidimensional de tip *long*;
- al treilea este de tip *double*.

4. void c(int,long[],double);

Acest prototip exprimă acelaşi lucru cu cel precedent.

Compilatorul utilizează datele din prototip pentru a verifica tipurile parametrilor de la apel (parametri efectivi). În cazul în care un parametru efectiv are un tip diferit de tipul corespunzător din prototip, compilatorul C converteşte automat valoarea parametrului efectiv spre tipul indicat în prototip.

Utilizatorii limbajelor C şi C++ pot folosi o serie de funcţii aflate în bibliotecile standard ale acestor limbaje. Apelul unei funcţii de bibliotecă implică şi el prezenţa prealabilă a prototipului funcţiei respective în textul sursă. Pentru a simplifica inserarea în textul sursă a prototipurilor funcţiilor de bibliotecă, s-au construit fişiere cu astfel de prototipuri. Ele au extensia .h (header). Un astfel de fişier conţine prototipuri pentru funcţii de bibliotecă "înrudite". De exemplu, fişierul *stdio.h* conţine prototipuri pentru funcţiile de bibliotecă utilizate frecvent în operaţii de intrare/ieşire, fişierul *string.h* conţine prototipurile pentru funcţiile utilizate la prelucrarea şirurilor de caractere etc.

Menţionăm că mediile integrate de dezvoltare Turbo C şi C++ permit utilizatorului să găsească prototipurile funcţiilor de bibliotecă. În acest scop se procedează astfel:

- se tastează numele funcţiei pentru care se doreşte să se găsească

informaţii;

- se fixează cursorul pe o literă arbitrară a numelui funcţiei;
- se tastează <CTRL>-F1.

Se obţine prototipul funcţiei, precum şi fişierul cu extensia *.h* care-l conţine. De asemenea, se afişează şi alte informaţii în legătură cu funcţia respectivă, utile pentru apelurile ei.

1.12. Preprocesare

Un program sursă C sau C++ poate fi prelucrat înainte de a fi supus compilării. O astfel de prelucrare se numeşte *preprocesare*. Ea este realizată automat înaintea compilării. Preprocesarea constă, în principiu, în substituţii. Preprocesarea asigură:

- includeri de fişiere cu texte sursă;
- definiţii şi apeluri de macrouri;
- compilare condiţionată.

Preprocesarea se realizează prin prelucrarea unor informaţii specifice care au ca prim caracter caracterul diez (#).

În paragraful de faţă vom aborda construcţia *#include* şi parţial construcţia *#define* utilizată la substituiri de succesiuni de caractere. Ulterior se vor descrie şi alte facilităţi oferite de preprocesare.

1.12.1. Includeri de fişiere cu texte sursă

Un fişier cu text sursă poate fi inclus cu ajutorul construcţiei *#include*. Această construcţie are unul din următoarele formate:

#include "*specificator_de_fişier*"

sau

#include <*specificator_de_fişier*>

unde:

specificator_de_fişier - Depinde de sistemul de operare. El defineşte un fişier cu text sursă păstrat pe disc. În faza de preprocesare, textul fişierului respectiv se substituie construcţiei *#include*. În felul acesta textul fişierului respectiv ia parte la compilare împreună cu textul în care a fost inclus.

În cazul sistemului de operare DOS, specificatorul de fişier trebuie să fie

un nume de fişier împreună cu extensia lui (.C pentru compilatorul C, .CPP pentru compilatorul C++, .H pentru fişiere de tip header (fişiere cu prototipuri etc.). De asemenea, în afară de numele şi extensia fişierului, specificatorul de fişier poate conţine şi o "cale", dacă este necesar, pentru localizarea fişierului.

Diferenţa dintre cele două formate constă în modul de căutare al fişierului de inclus.

Formatul cu parantezele unghiulare <...> se utilizează la includerea *fişierelor standard*, cum sînt cele care conţin prototipuri pentru funcţiile de bibliotecă. Directoarele în care se caută aceste fişiere se definesc în prealabil cu ajutorul submeniului *Directories* al meniului *Options* din mediul integrat de dezvoltare Turbo C sau Turbo C++. Ordinea de căutare în aceste directoare corespunde cu ordinea în care au fost definite aceste directoare. În cazul în care fişierul căutat nu este găsit în aceste directoare, se dă un mesaj de eroare.

În cazul în care se utilizează caracterele *ghilimele*, fişierul se caută în directorul curent sau conform "căii" dacă aceasta este prezentă.

Un fişier standard care se include frecvent este fişierul *stdio.h* care conţine prototipurile pentru o serie de funcţii ce realizează operaţii de intrare/ieşire. El se include folosind construcţia:

#include <stdio.h>

Exemple:

1. #include "fis1.cpp"

   Se include textul fişierului fis1.cpp aflat în directorul curent.

2. #include "c:\\tc\\sursă\\fis2.c"

   În acest exemplu este prezentă calea fişierului fis2.c. Se observă că pentru a descrie calea, caracterul backslash se dublează conform convenţiei de reprezentare a caracterului backslash într-un şir de caractere.

Un text inserat cu ajutorul construcţiei #include la rîndul său poate conţine construcţii #include pentru alte fişiere.

Construcţiile #include se scriu, de obicei, la începutul fişierelor sursă, pentru ca textele inserate să fie valabile în tot fişierul sursă care se compilează.

1.12.2. Substituiri de succesiuni de caractere la preprocesare

Construcţia *#define* se poate folosi la substituţii de succesiuni de caractere. În acest scop se utilizează formatul:

#define *nume succesiune_de_caractere*

unde:

nume - Este precedat şi urmat de cel puţin un spaţiu.

Folosind această construcţie, preprocesarea *substituie nume* cu *succesiune_de_caractere* peste tot în textul sursă care urmează construcţiei #define respective, exceptînd cazul cînd *nume* apare într-un şir de caractere sau într-un comentariu.

Numele definit printr-o construcţie *#define* substituindu-se prin secvenţa de caractere corespunzătoare nu mai este prezent la compilare.

De obicei, un astfel de nume se scrie cu litere mari pentru a scoate în evidenţă faptul că el este definit printr-o construcţie *#define.*

Succesiune_de_caractere începe cu primul caracter care nu este alb. Ea poate fi continuată pe mai multe linii terminînd rîndul care dorim să se continue cu backslash.

#define este folosită frecvent pentru a defini constante. De aceea, uneori un nume definit printr-o construcţie #define se spune că este o *constantă simbolică.*

O construcţie *#define* autorizează substituţia pe care o defineşte din punctul în care ea este scrisă şi pînă la sfîrşitul fişierului în care ea este scrisă sau pînă la întîlnirea unei construcţii *#undef* care o anulează. Aceasta are formatul:

#undef *nume*

La întîlnirea ei, se dezactivează substituirea lui nume cu succesiunea de caractere care i-a fost ataşată în prealabil printr-o construcţie *#define.*

Succesiunea de caractere dintr-o construcţie *#define* poate conţine nume care au fost definite în prealabil prin alte construcţii *#define.*

Exemple:

1.

```
...
#define A 100
/* A se substitue prin 100 incepind din acest punct al fisierului sursa */
```

```
...
#define FACT 20
#define DIMMAX (A*FACT)
...
char tab[DIMMAX];
...
double mat[A][FACT];
...
```

După preprocesare se obţin declaraţiile:

```
...
char tab[(100*20)];
...
double mat[100][20];
...
```

DIMMAX s-a substituit prin (100*20) şi nu prin valoarea 2000 a produsului deoarece preprocesorul nu face calcule ci numai substituţii. Din această cauză se recomandă ca expresiile utilizate în construcţiile #define să fie incluse în paranteze rotunde.

2.

```
#define A 123
#define B A+120
...
x=3*B; // se substitue prin x=3*123+120;
...
```

3.

```
#define A 123
#define B (A+120)
...
x=3*B; // se substitue prin x=3*(123+120);
...
```

4.

```
...
#define A 100
...
int x[A+1]; // declaratia devine int x[100+1];
...
#undef A
#define A 3.5
```

```
...
double y;
...
y=A; // instructiunea devine y=3.5;
...
```

În exemplele de mai sus *A* este o constantă simbolică. Ea este un nume atribuit unei constante, nume care însă nu mai este prezent în faza de compilare.

B defineşte o expresie constantă, care este evaluată de compilator la întîlnirea ei.

Constantele simbolice se utilizează frecvent în locul constantelor obişnuite deoarece ele prezintă următoarele avantaje:

- Permite să se atribuie nume sugestive unor constante. Este mult mai sugestiv să folosim constanta simbolică *PI* definită prin:

  #define PI 3.14159

  decît valoarea ei.

- Permite realizarea de prescurtări. De exemplu, dacă valoarea 3.14159 se foloseşte de mai multe ori în program, atunci este mai simplu să folosim numele *PI* atribuit ei.
- Permite înlocuirea simplă a unei constante printr-o alta. Dacă o constantă se foloseşte de mai multe ori într-un program şi ulterior se constată că valoarea ei trebuie schimbată (de exemplu este o constantă flotantă căreia trebuie să-i schimbăm precizia), atunci este mult mai simplu să-i schimbăm valoarea o singură dată în construcţia *#define*, decît să o căutăm peste tot în program şi să-i schimbăm valoarea în mod corespunzător.

2. INTRĂRI/IEŞIRI STANDARD

Prin *intrări/ieşiri* înţelegem un set de operaţii care permit schimbul de date între un program şi un periferic.

În general, operaţia de introducere a datelor de la un periferic se numeşte *citire*, iar cea de ieşire pe un periferic *scriere*.

Numim *terminal standard* terminalul de la care s-a lansat programul. De obicei, terminalele standard sînt de tip *display*. În acest caz operaţia de scriere se mai numeşte şi *afişare*.

Limbajul C nu dispune de instrucţiuni specifice pentru operaţiile de intrare/ieşire. Ele pot fi realizate apelînd *funcţii* construite special pentru acest scop. Astfel de funcţii există în biblioteca limbajului C. Operaţiile de intrare/ieşire realizate prin intermediul funcţiilor de bibliotecă le numim *operaţii de intrare/ieşire standard.*

Utilizatorul poate el însuşi să construiască astfel de funcţii, în cazul în care nu sînt utilizabile cele standard existente.

Funcţiile de bibliotecă, utilizate mai frecvent pentru realizarea operaţiilor de intrare/ieşire folosind terminalul standard sînt:

- pentru intrări: *getch*, *getche*, *gets* şi *scanf*;
- pentru ieşiri: *putch*, *puts* şi *printf*.

La acestea mai adăugăm macrourile:

getchar - pentru intrări;

şi

putchar - pentru ieşiri.

Aceste macrouri sînt definite în fişierul *stdio.h* şi din această cauză utilizarea lor implică includerea acestui fişier.

2.1. Funcţiile getch şi getche

Funcţiile *getch* şi *getche* sînt dependente de implementare. Mai jos indicăm utilizările lor sub mediile de programare Turbo C şi Turbo C++.

Ambele funcţii permit citirea direct de la tastatură a unui caracter.

Funcţia *getch* citeşte de la tastatură *fără ecou*, deci caracterul tastat la display nu se afişează pe ecranul acestuia. Ea permite citirea caracterelor de

la tastatură atît a celor corespunzătoare codului ASCII, cît şi a celor corespunzătoare unor funcţii speciale cum ar fi tastele F1, F2 etc. sau combinaţii de taste speciale.

La citirea unui caracter al codului ASCII, funcţia returnează codul ASCII al caracterului respectiv.

În cazul în care se acţionează o tastă care nu corespunde unui caracter ASCII, funcţia *getch* se apelează de *două ori:* la primul apel funcţia returnează valoarea *zero*, iar la cel de al doilea apel se returnează o valoare specifică tastei acţionate.

Funcţia *getche* este analogă cu funcţia *getch*, cu singura diferenţă că ea realizează citirea *cu ecou* a caracterului tastat. Aceasta înseamnă că se afişează automat pe ecranul terminalului caracterul tastat.

Ambele funcţii *nu au parametri* şi se pot apela ca operanzi în expresii.

La apelarea lor se vizualizează *fereastra utilizator* şi se aşteaptă tastarea unui caracter. Programul continuă după tastarea caracterului.

Un apel de forma:

getch();

se utilizează în cazul în care dorim să vizualizăm fereastra utilizator şi să blocăm programul pentru a analiza conţinutul curent al ecranului. Pentru a debloca programul se acţionează o tastă corespunzătoare unui caracter al codului ASCII. Caracterul respectiv se citeşte fără ecou şi apoi se continuă execuţia programului.

Funcţiile *getch* şi *getche* au prototipurile în fişierul *conio.h*, deci utilizarea lor implică includerea acestui fişier.

2.2. Funcţia putch

Funcţia *putch* afişează un caracter pe ecranul terminalului standard. Ea are un parametru care determină imaginea afişată la terminal.

Funcţia *putch* poate fi apelată astfel:

putch(*expresie*);

Prin acest apel se afişează imaginea definită de valoarea parametrului *expresie*. Valoarea parametrului se interpretează ca fiind codul ASCII al caracterului care se afişează.

Dacă valoarea expresiei se află în intervalul [32,126], atunci se afişează un caracter imprimabil al codului ASCII. Dacă valoarea respectivă este în

afara acestui interval, atunci se afişează diferite imagini care pot fi folosite în diverse scopuri, cum poate fi de exemplu trasarea de chenare.

Funcţia *putch* afişează caractere colorate în conformitate cu culoarea curentă setată în modul *text* de funcţionare al ecranului.

La revenirea din funcţia *putch* se returnează codul imaginii afişate (valoarea parametrului de la apel).

Prototipul funcţiei *putch* se află în fişierul *conio.h*.

Exerciţii:

2.1 Să se scrie un program care citeşte un caracter imprimabil şi-l afişează apoi pe ecran.

Caracterul respectiv se citeşte folosind apelul:

getch()

La revenirea din getch() se returnează codul ASCII al caracterului tastat. Acest cod urmează a fi folosit la apelul funcţiei *putch* pentru a afişa caracterul tastat. Deci pentru a rezolva problema de faţă putem folosi construcţia:

putch(getch());

În acest caz parametrul efectiv este o expresie formată dintr-un singur operand, operand care constă din apelul funcţiei *getch*. Aceasta este posibil deoarece orice funcţie care returnează o valoare poate fi apelată ca operand al unei expresii.

Menţionăm că funcţia *getch* returnează valoarea 13 (CR) la citirea caracterului rezultat din acţionarea tastei *Enter*.

PROGRAMUL BII1

```
#include <conio.h>
main() /* citeste fara ecou un caracter imprimabil ASCII
          si-l afiseaza pe ecran */
{
 putch(getch());
}
```

Observaţii:

1. Programul este inefectiv dacă se acţionează o tastă ce nu corespunde caracterelor ASCII.

2. Caracterul afişat prin putch se poate vedea în fereastra utilizator. Aceasta se vizualizează tastînd:

<Alt>-F5

Se revine în fereastra utilizator acţionînd o tastă oarecare.

2.2 Să se scrie un program care citeşte fără ecou un caracter imprimabil ASCII, îl afişează la terminal şi apoi trece cursorul pe linia următoare.

Programul de faţă este asemănător cu cel precedent. În plus se cere trecerea cursorului pe linia următoare. În acest scop se apelează funcţia *putch* cu codul caracterului *newline*:

putch(10);

Menţionăm că putem înlocui valoarea 10 prin constanta caracter '\n':

putch ('\n');

PROGRAMUL BII2

```
#include <conio.h>
main() /* citeste fara ecou un caracter imprimabil ASCII, il afiseaza la
          terminal si apoi se trece cursorul pe linia urmatoare */
{
 putch(getch());
 putch('\n');
}
```

2.3 Programul BII3 realizează acelaşi lucru ca şi programul BII2, cu deosebirea că înainte de a se termina execuţia lui, se afişează fereastra utilizator şi se aşteaptă acţionarea unei taste.

PROGRAMUL BII3

```
#include <conio.h>
main() /* citeste fara ecou un caracter imprimabil ASCII, il afiseaza,
          trece cursorul pe linia urmatoare, afiseaza fereastra uti-
          lizator si se asteapta actionarea unei taste; programul se ter-
          mina dupa actionarea unei taste */
{
 putch(getch());
 putch('\n');
 getch();
}
```

2.3. Macrourile getchar şi putchar

Aceste macrouri sînt definite în fişierul *stdio.h*. Ele se apelează la fel ca funcţiile.

Macroul *getchar* permite citirea cu ecou a caracterelor de la terminalul standard. Se pot citi numai caractere ale codului ASCII, nu şi caractere corespunzătoare tastelor speciale. Prin intermediul macroului *getchar* caracterele nu se citesc direct de la tastatură. Caracterele tastate la terminal se introduc într-o zonă tampon pînă la acţionarea tastei *Enter*. În acest moment, în zona tampon, se introduce caracterul de rînd nou (newline) şi se continuă execuţia lui *getchar*. Se revine din *getchar* returnîndu-se codul ASCII al caracterului curent din zona tampon. La un nou apel al lui getchar se revine cu codul ASCII al caracterului următor din zona tampon. La epuizarea tuturor caracterelor din zona tampon, apelul lui *getchar* implică tastarea la terminal a unui nou set de caractere care se reîncarcă în zona tampon.

Un astfel de mod de desfăşurare a operaţiei de citire implică o anumită *organizare* a memoriei şi accesului la caractere, organizare care conduce la noţiunea de *fişier*.

În general, prin fişier se înţelege o mulţime ordonată de elemente păstrate pe suporturi. Elementele unui fişier se numesc *înregistrări*. În mod frecvent fişierele sînt păstrate pe discuri. Cu toate acestea, este util să se considere organizate în fişiere chiar şi datele care se tastează sau se afişează la terminal. În acest caz înregistrarea este un rînd afişat la terminal sau succesiunea de caractere tastată la terminal şi terminată prin acţionarea tastei Enter.

Fişierele conţin o înregistrare specială care marchează *sfîrşitul de fişier*. Această înregistrare se realizează la tastatură prin secvenţe speciale. În cazul limbajelor Turbo C şi Turbo C++, sfîrşitul de fişier se obţine tastînd:

<CTRL>-Z

al cărui ecou este:

^Z

Macroul *getchar* returnează valoarea constantei simbolice EOF (End Of File) la întîlnirea sfîrşitului de fişier. Această constantă este definită în fişierul *stdio.h*. Valoarea ei depinde de implementare. În cazul mediilor de programare Turbo C şi Turbo C++, ea are valoarea -1.

Macroul *getchar* se apelează fără parametri şi de obicei este un operand al unei expresii: **getchar()**

Macroul *putchar* afişează un caracter al codului ASCII. El returnează codul caracterului afişat sau -1 la eroare.

Se poate apela prin formatul de mai jos:

putchar(*expresie*);

Valoarea expresiei reprezintă codul ASCII al caracterului care se afişează.

Observaţii:

1. Citirea caracterelor prin intermediul zonelor tampon are avantajul că permite corectarea erorilor la tastare. Astfel, operatorul de la terminal poate să modifice o secvenţă de caractere tastate, înainte de a acţiona tasta Enter.

2. Macroul *getchar* returnează valoarea 10 (newline) la citirea caracterului corespunzător acţionării tastei Enter.

3. Apelul

   putchar(10);

   sau echivalentul său

   putchar('\n');

   are ca efect trecerea cursorului în coloana unu de pe linia următoare.

Exerciţii:

2.4 Să se scrie un program care citeşte un caracter folosind macroul *getchar*, îl afişează folosind macroul *putchar*, trece cursorul în coloana unu din linia următoare şi apoi afişează fereastra utilizator pînă la acţionarea unei taste.

Acest exerciţiu este analog cu 2.3. În acest caz se cere schimbarea funcţiilor *getch* şi *putch* cu macrourile *getchar* şi respectiv *putchar*.

PROGRAMUL BII4

```
#include <stdio.h>
#include <conio.h>

main() /* - citeste un caracter folosind getchar;
          - afiseaza caracterul folosind putchar;
          - trece cursorul la inceputul rindului urmator;
          - blocheaza programul pina la tastarea unui caracter */
```

```
{
 putchar(getchar());
 putchar('\n');
 getch();
}
```

Observaţie:

Deoarece se foloseşte macroul *getchar*, operatorul va tasta un caracter ASCII care trebuie să fie urmat de acţionarea tastei Enter. Fără acţionarea tastei Enter, programul rămîne în aşteptare. După acţionarea tastei Enter se va reveni din *getchar* cu codul ASCII al primului caracter tastat.

Dacă prima tastă acţionată este chiar tasta Enter, atunci programul continuă execuţia în mod automat şi se revine din *getchar* cu valoarea 10 (valoarea corespunzătoare tastei Enter).

2.4. Funcţiile gets şi puts

Funcţia *gets* poate fi folosită pentru a *introduce* de la terminalul standard o succesiune de caractere terminată prin acţionarea tastei Enter.

Citirea se face *cu ecou*. Se pot citi numai caractere ale codului ASCII.

Funcţia are ca parametru *adresa de început* a zonei de memorie în care se păstrează caracterele citite. De obicei, această zonă de memorie este alocată unui tablou unidimensional de tip *char*.

Deoarece numele unui tablou are ca valoare adresa de început a zonei de memorie alocate, rezultă că numele unui tablou poate fi utilizat ca parametru în apelul funcţiei *gets*. În felul acesta, caracterele citite se vor păstra în tabloul respectiv.

Funcţia *gets* returnează adresa de început a zonei de memorie în care s-au păstrat caracterele.

La întîlnirea sfîrşitului de fişier (<CTRL>-Z) se returnează valoarea zero. Zero nu reprezintă o adresă posibilă pentru gets şi de aceea, ea poate fi folosită pentru a semnala sfîrşitul de fişier. De obicei, valoarea returnată de *gets* nu se testează faţă de zero, ci faţă de constanta simbolică *NULL* definită în fişierul *stdio.h*.

Caracterul *newline*, care termină rîndul, nu se păstrează în memorie. În locul lui se păstrează caracterul *NUL* ('\0'). În felul acesta, caracterele citite prin *gets* formează un şir de caractere în conformitate cu cerinţele limbajelor C şi C++ (secvenţa de caractere se termină prin caracterul *NUL*).

Din cele de mai sus rezultă că dacă *tab* este declarat prin:

char tab[255];

atunci apelul:

gets(tab);

păstrează în *tab* succesiunea de caractere tastată la terminalul standard în linia curentă. Totodată, *newline* se înlocuieşte cu *NUL*.

Funcţia *puts* afişează la terminalul standard un şir de caractere ale codului ASCII. Cursorul, după afişarea şirului respectiv, se trece automat în coloana întîi de pe linia următoare (deci caracterul *NUL* se înlocuieşte cu newline).

Funcţia are ca parametru adresa de început a zonei de memorie care conţine caracterele de afişat.

În cazul în care şirul de caractere care se afişează se păstrează într-un tablou unidimensional de tip *char*, drept parametru se poate folosi numele acestui tablou.

Funcţia *puts* returnează codul ultimului caracter al şirului de caractere afişat (caracterul care precede pe *NUL*) sau -1 la eroare.

Dacă *tab* are declaraţia de mai sus şi el păstrează un şir de caractere, atunci apelul

puts(tab);

afişează la terminalul standard şirul respectiv de caractere şi apoi trece cursorul în coloana întîi de pe rîndul următor.

Funcţiile *gets* şi *puts* au prototipurile în fişierul stdio.h.

Exerciţii:

2.5 Să se scrie un program care citeşte de la intrarea standard numele şi prenumele unei persoane, afişează iniţialele persoanei respective pe un rînd, fiecare iniţială este urmată de un punct.

Numele şi prenumele se tastează pe două rînduri diferite şi apoi se vor afişa iniţialele respective. În final se trece cursorul în coloana întîi a liniei următoare şi se afişează fereastra utilizator.

PROGRAMUL BII5

```
#include <stdio.h>
#include <conio.h>
```

```
main() /*- citeste numele si prenumele unei persoane tastate
            pe doua linii separate;
          - afiseaza pe un rind initialele urmate de punct;
          - afiseaza fereastra utilizator. */
{
 char nume[255];
 char prenume[255];

 gets(nume);           //citeste prima linie tastata la terminal
 gets(prenume);        //citeste linia a doua tasta la terminal
 putchar(nume[0]);            //afiseaza prima initiala
 putchar('.');                //afiseaza punct dupa prima initiala
 putchar(prenume[0]);  //afiseaza a doua initiala
 putchar('.');         //punct dupa a doua initiala
 putchar('\n');        //trece in coloana intii din linia urmatoare
 getch();              //se afiseaza fereastra utilizator; pentru a termina
                       // programul se actioneaza o tasta oarecare
}
```

2.6 Să se modifice programul precedent aşa încît după afişarea iniţialelor să se afişeze textul:

Pentru a termina actionati o tasta

Acest text se poate afişa cu ajutorul funcţiei *puts*, pe care o apelăm înainte de apelul lui *getch*. Se poate utiliza următorul apel al funcţiei *puts*:

puts("Pentru a termina actionati o tasta");

În acest apel parametrul efectiv este o expresie redusă la un operand care este o constantă şir.

La acest apel compilatorul păstrează caracterele constantei şir într-o zonă de memorie şi apoi realizează apelul funcţiei puts folosind adresa de început a zonei respective.

PROGRAMUL BII6

```
#include <stdio.h>
#include <conio.h>

main() /* - citeste numele si prenumele unei persoane tastate pe doua
            linii separate;
          - afiseaza initialele persoanei urmate de puncte pe un acelasi
            rind;
```

- pe rindul urmator se afiseaza textul:
  Pentru a termina programul actionati o tasta;
- se afiseaza fereastra utilizator. */

```
{
 char nume[255];
 char prenume[255];

 gets(nume);
 gets(prenume);
 putchar(nume[0]);
 putchar('.');
 putchar(prenume[0]);
 putchar('.');
 puts("\nPentru a termina programul actionati\
        o tasta");
 getch();
}
```

2.5. Funcţia printf

Funcţia *printf* poate fi folosită pentru a afişa date pe ecranul terminalului standard sub controlul unor formate. Ea poate fi apelată printr-o construcţie de forma:

printf(*control,par1,par2,...,parn*);

unde:

control - Este un şir de caractere care defineşte textele şi formatele datelor care se scriu;

par1,par2,...,parn - Sînt expresii.
- Valorile lor se scriu conform specificatorilor de format prezenţi în parametrul de control.

Datele gestionate prin *printf* sînt supuse unor transformări. Aceasta din cauză că datele au un format extern şi unul intern.

Parametrul *control* al funcţiei *printf* defineşte aceste transformări ale datelor care se afişează pe ecran, transformări care se mai numesc *conversii*. El conţine aşa numiţii *specificatori de format* care definesc conversiile datelor din format intern în format extern. În afara acestor specificatori de format, parametrul *control* al funcţiei *printf* poate conţine succesiuni de caractere care se afişează ca atare în poziţii corespunzătoare.

În concluzie, parametrul *control* are în compunerea sa:

- succesiuni de caractere care se afişează ca atare;
- specificatori de format care definesc conversiile valorilor parametrilor par1, par2, .., parn din format intern în format extern.

Menţionăm că aceste construcţii nu sînt totdeauna ambele prezente. De exemplu, dacă dorim să se afişeze un text, atunci parametrul *control* nu conţine specificatorii de format, decît numai textul respectiv, iar parametrii *par1, par2, ..., parn* sînt absenţi. În mod analog, parametrul *control* poate să conţină numai specificatori de format.

În mod frecvent, parametrul *control* conţine atît specificatori de format, cît şi alte caractere.

Exemplu:

Apelul funcţiei *puts* folosit în exerciţiul 2.6.:

```
puts("\nPentru a termina programul actionati\
       o tasta");
```

poate fi înlocuit printr-un apel al funcţiei *printf* care conţine numai parametrul de *control* fără specificatori de format:

```
printf("\nPentru a termina programul actionati\
        o tasta\n");
```

În cazul apelului funcţiei *printf* s-a folosit caracterul *newline* de la sfîrşitul şirului de caractere pentru ca după afişarea textului, cursorul să treacă în coloana unu din linia următoare. Acest fapt se realizează automat în cazul funcţiei *puts* şi deci la apelul ei şirul de caractere nu se mai termină cu *newline*.

Specificatorii de format, dacă există, se corespund cu parametrii *par1, par2, ..., parn*. Astfel, al k-lea specificator de format controlează afişarea valorii parametrului *park*.

Formatul extern al unei date constă dintr-o succesiune de caractere imprimabile. Ele se afişează într-o zonă numită *cîmp*.

Un specificator de format începe cu un caracter procent (%). În continuare, mai pot exista caracterele indicate mai jos:

un caracter minus opţional	- Implicit datele se cadrează în dreapta cîmpului în care ele se scriu. Dacă este prezent caracterul minus, atunci data corespunzătoare lui este cadrată în stînga.
un şir de cifre zecimale opţional	- Care defineşte dimensiunea minimă a cîmpului în care se afişează caracterele care intră în compunerea formatului extern al datei. În cazul în care data necesită un cîmp de

dimensiune mai mare decît cel precizat, ea se va scrie pe atîtea caractere cîte îi sînt necesare. În cazul în care data necesită un cîmp mai mic decît cel definit în specificatorul de format, ea se va scrie în cîmpul respectiv în dreapta sau în stînga, dacă este prezent caracterul minus în specificatorul de format corespunzător ei. De asemenea, în acest caz cîmpul se completează cu caractere nesemnificative; implicit, caracterele nesemnificative sînt spaţii. Caracterele nesemnificative vor fi zerouri dacă numărul ce indică dimensiunea minimă a cîmpului începe cu un zero nesemnificativ (aici zeroul nesemnificativ nu înseamnă că numărul este în octal, deoarece acest şir de cifre nu reprezintă o constantă întreagă, ci are semnificaţia amintită mai sus).

un punct opţional, urmat de un şir de cifre zecimale - Şirul de cifre zecimale aflate după punct defineşte precizia datei care se scrie sub controlul specificatorului respectiv. Dacă punctul este prezent, atunci şi precizia este prezentă. În cazul în care data care se scrie este flotantă, precizia defineşte numărul de zecimale care se scriu. În cazul în care data este un şir de caractere, precizia indică numărul de caractere care se scriu.

una sau două litere - Care definesc tipul de conversie aplicat datei care se scrie.

Din cele de mai sus, rezultă că un specificator de format începe cu un caracter procent şi se termină cu o literă.

Între caracterul procent şi litera care sfîrşeşte un specificator de format mai pot apare şi alte caractere a căror interpretare s-a dat mai sus.

În cazul în care după caracterul procent urmează un caracter diferit de cele folosite la definirea specificatorilor de format, caracterul procent respectiv se ignoră şi se afişează caracterul care urmează după el. Această regulă permite afişarea însuşi a caracterului procent. Astfel, succesiunea:

%%

nu reprezintă un specificator de format, primul caracter procent se ignoră, iar cel de al doilea caracter procent se va afişa la terminal.

Funcţia *printf* returnează numărul de octeţi (caractere) care se afişează la terminal sau -1 la eroare.

Prototipul funcţiei *printf* se află în fişierul *stdio.h*.

În cele ce urmează vom indica principalele conversii realizate prin

specificatorii de format. Ele sînt definite de literele de la sfîrşitul specificatorului de format.

2.5.1. Litera c

Permite afişarea unui caracter. Valoarea parametrului corespunzător este interpretată ca fiind codul ASCII al unui caracter şi caracterul respectiv se afişează în cîmpul definit de specificator.

2.5.2. Litera s

Permite afişarea unui şir de caractere.

Parametrul corespunzător are ca valoare adresa de început a zonei de memorie care conţine caracterele şirului de afişat. Se afişează toate caracterele şirului pînă la întîlnirea caracterului NUL.

Exemple:

1. Secvenţa de apeluri:

```
putchar(getchar());
```

se poate realiza folosind funcţia *printf* astfel:

```
printf("%c",getchar());
```

2. Secvenţa de apeluri:

```
putchar(getchar());
putchar('\n');
```

se poate înlocui prin:

```
printf("%c\n",getchar());
```

3. Apelul lui *printf* pentru a afişa un şir de caractere:

```
printf("abc");
```

se poate scrie folosind specificatorul de format pentru şiruri de caractere:

```
printf("%s","abc");
```

În ambele cazuri, compilatorul păstrează şirul de caractere "abc" într-o zonă specială prevăzută pentru acest fapt şi apelează funcţia *printf* folosind ca parametru adresa de început a acestei zone de memorie.

4. Fie apelul lui *printf:*

```
printf("*%4c*",getchar());
```

Caracterul citit prin *getchar* se afişează într-un cîmp de 4 caractere, cadrat în dreapta. De exemplu, dacă s-a citit caracterul *A*, atunci apelul de mai sus va afişa:

```
*   A*
```

5. Apelul:

```
printf("*%-4c*",getchar());
```

va afişa caracterul citit într-un cîmp de 4 caractere, cadrat în stînga. De exemplu, dacă se citeşte caracterul *A*, atunci apelul de mai sus va afişa:

```
*A   *
```

6. Fie apelul:

```
printf("*%10s*","abc");
```

se afişează:

```
*       abc*
```

7. Apelul:

```
printf("*%-10s*", "abc");
```

afişează:

```
*abc       *
```

8. Apelul:

```
printf("*%15.10*","limbajul C++");
```

afişează:

```
*     limbajul C*
```

2.5.3. Litera d

Datele de tip *int* pot fi convertite şi afişate în zecimal folosind un specificator de format terminat cu litera *d*. Valorile negative se afişează precedate de semnul minus.

Exemple:

1. Specificatorul

   %d

   se foloseşte pentru a afişa, în zecimal, o dată de tip *int*.

2. Apelul:

   printf("*%10d*",123);

   afişează:

   *   123*

3. Apelul:

   printf("*%-10d*",123);

   afişează:

   *123   *

4. Apelul:

   printf("*%010d*",123);

   afişează:

   0000000123

2.5.4. Litera o

Datele de tip *int* sau *unsigned* pot fi convertite şi afişate în octal folosind un specificator de format terminat cu litera *o*.

Fiecare grupă de trei cifre binare se înlocuieşte printr-o cifră octală. Gruparea cifrelor se face de la dreapta spre stînga, adică de la ordinele inferioare spre cele superioare.

Exemplu:

Apelul:

printf("*%10o*",123);

afişează:

*   173*

2.5.5. Literele x şi X

Datele de tip *int* sau *unsigned* pot fi convertite şi afişate în hexazecimal folosind un specificator de format terminat cu litera *x* sau *X*. În acest scop, patru cifre binare se înlocuiesc printr-o cifră hexazecimală.

Exemplu:

Apelul:

```
printf("*%10x*",123);
```

afişează:

```
*        7b*
```

În cazul în care se foloseşte litera mare *X*, cifrele hexa peste 9 se afişează cu litere mari.

2.5.6. Litera u

Litera *u* este asemănătoare cu litera *d*. În acest caz se realizează conversia unei date binare de tip *unsigned* în zecimal.

2.5.7. Litera l

Litera *l* poate precede pe una din literele *d, o, x, X* sau *u*.

În acest caz se fac conversii din tipul *long* sau *long unsigned*. Astfel, specificatorul terminat în *ld* realizează conversia din *long* în zecimal. Specificatorul *lu* realizează conversia din *long unsigned* în zecimal. Specificatorul *lo* face conversie din *long* sau *long unsigned* în octal. Specificatorul *lx* sau *lX* face conversia din *long* sau *long unsigned* în hexazecimal.

2.5.8. Litera f

Litera *f* permite conversia datelor de tip *float* sau *double* spre formatele:

parte_întreagă.parte_fracţionară

sau

parte_întreagă

Partea întreagă este un şir de cifre zecimale care este precedat de semnul minus dacă numărul este negativ.

Numărul zecimalelor se defineşte de precizia indicată în specificatorul

de format. Dacă ea este absentă, atunci se afişează 6 zecimale. Ultima cifră afişată este rotunjită. Rotunjirea se face prin adaus dacă prima cifră neglijată este cel puţin egală cu 5 şi prin lipsă în caz contrar.

Exemple:

Valoarea datei	Specificator	Afişare
3.14159265	%5f	3.141593
123.672	%7f	123.672000
3.14159265	%7.2f	3.14
123.672	%10.1f	123.7
-123.672	%10.1f	-123.7
3.14159265	%10.0f	3
123.672	%10.0f	124

2.5.9. Literele e şi E

Aceste litere permit conversia datelor flotante de tip *float* sau *double* spre formatele:

parte_întreagă.parte_fracţionară exponent

sau

parte_întreagă exponent

Partea întreagă este o cifră zecimală care este precedată de semnul minus dacă numărul este negativ.

Numărul zecimalelor este cel definit de precizie dacă aceasta este prezentă în specificatorul de format. În cazul în care precizia este absentă, numărul zecimalelor afişate este egal cu 6. Ultima cifră afişată este rotunjită conform regulei indicate mai sus la descrierea literei *f*.

Exponentul începe cu litera *e* dacă specificatorul de format se termină cu *e* şi cu *E* dacă el se termină cu *E*. Urmează un semn plus sau minus dacă acesta este negativ. După semn se află un întreg zecimal de cel puţin 2 cifre.

Exponentul defineşte o putere a lui zece care înmulţeşte restul reprezentării numărului pentru a obţine valoarea reală a acestuia.

Exemple:

Valoarea datei	Specificator	Afişare
3.14159265	%e	3.141593e+00
123.672	%e	1.236720e+02
123.672	%.1E	1.2E+02

0.673	%E	6.730000E-01
123.672	%.0E	1E+02

2.5.10. Literele g şi G

Specificatorul de format terminat cu litera *g* sau *G* afişează data respectivă fie ca în cazul specificatorului terminat cu *f*, fie ca în cazul specificatorului terminat cu *e* sau *E*. Formatul de afişare cu exponent sau fără, se alege automat în aşa fel încît afişarea să ocupe un număr minim de caractere. De asemenea, specificatorii %g şi %G afişează 6 zecimale numai dacă acestea sînt toate semnificative.

Se foloseşte litera *e* la scrierea exponentului dacă specificatorul se termină cu *g* şi *E* pentru specificatorul terminat cu *G*.

Observaţie:

Datele de tip *float* au o precizie de 6-7 zecimale. De aceea, pentru datele de acest tip nu se recomandă să se afişeze un număr mai mare de zecimale.

Datele de tip *double* au o precizie de pînă la 15 zecimale şi deci, în acest caz, putem afişa pînă la 15-16 zecímale. Afişarea unui număr mai mare de zecimale nu are sens.

2.5.11. Litera L

Litera *L* poate precede una din literele *f, e, E, g* şi *G*. În acest caz data care se afişează este de tip *long double*. Convenţiile de afişare sînt aceleaşi, adică datele afişate cu un specificator de format terminat cu *Lf* sînt fără exponent, cele cu un specificator de format terminat cu *Le* sau *LE* se afişează cu exponent, iar cele afişate cu un specificator de format cu *Lg* sau *LG* se afişează cu unul din formatele precedente care asigură o afişare cu un număr minim de caractere.

Exerciţii:

2.7 Să se scrie un program care afişează pe o linie, între două caractere asterisc, caracterul citit de la terminal prin *getchar*, apoi trece cursorul în coloana întîi de pe linia următoare şi afişează ecranul utilizator.

PROGRAMUL BII7

```
#include <stdio.h>
#include <conio.h>

main() /*- citeste un caracter cu getchar;
```

```
            - afiseaza pe o linie caracterul citit, intre doua
              caractere asterisc;
            - trece cursorul in coloana intii din linia urmatoare;
            - afiseaza fereastra utilizator. */
{
 printf("*%c*\n",getchar() );
 puts(" Actionati o tasta pentru a continua");
 getch();
}
```

2.8 Să se modifice programul din exemplul precedent în aşa fel încît caracterul citit prin *getchar* să se afişeze între două caractere procent.

PROGRAMUL BII8

```
#include <stdio.h>
#include <conio.h>

main() /* - citeste un caracter cu getchar;
          - afiseaza pe o linie caracterul citit, intre doua
            caractere procent;
          - trece cursorul in coloana intii din linia urmatoare;
          - afiseaza fereastra utilizator. */
{
 printf("%%%c%%\n",getchar() );
 puts(" Actionati o tasta pentru a continua");
 getch();
}
```

2.9 Programul de mai jos afişează textul "Limbajul C++", folosind următorii specificatori de format:

%s, %15s, %-15s, %10s, %15.10s şi %-15.10s

Pentru a pune în evidenţă cîmpul afişat prin specificatorii respectivi, vom include de fiecare dată cîmpul între caractere asterisc.

PROGRAMUL BII9

```
#include <stdio.h>
#include <conio.h>

#define V "Limbajul C++"
```

```
main() /* afiseaza, intre caractere asterisc, textul Limbajul C++
         folosind diferiti specificatori de format */

 {
  printf("*%s*\n", V);
  printf("*%15s*\n", V);
  printf("*%-15s*\n", V);
  printf("*%10s*\n", V);
  printf("*%15.10s*\n", V);
  printf("*%-15.10s*\n", V);
  puts (" Actionati o tasta pentru a continua");
  getch();
 }
```

Rezultatele execuţiei programului BII9 sînt:

```
*Limbajul C++*
*   Limbajul C++*
*Limbajul C++   *
*Limbajul C++*
*     Limbajul C*
*Limbajul C     *
Actionati o tasta pentru a continua
```

2.10 Să se scrie un program care citeşte un caracter cu *getchar* şi afişează codul său ASCII.

PROGRAMUL BII10

```
#include <stdio.h>
#include <conio.h>

main() /* citeste un caracter cu getchar si afiseaza codul sau ASCII */
{
 printf("%d\n", getchar() );
 puts( "Actionati o tasta pentru a continua");
 getch();
}
```

2.11 Să se scrie un program care citeşte un caracter care nu corespunde codului ASCII şi afişează codul său returnat prin funcţia *getch*.

Amintim că pentru a citi caractere care nu corespund codului ASCII,se apelează *getch* de două ori. La primul apel, *getch* returnează valoarea zero.

La cel de al doilea apel se returnează un cod specific caracterului citit.

PROGRAMUL BII11

```
#include <stdio.h>
#include <conio.h>

main() /* citeste un caracter care nu apartine codului ASCII si afiseaza
          codul caracterului citit */
{
 printf("%d\n", getch() ); //afiseaza 0
 printf("%d\n", getch() ); //afiseaza codul caracterului
 puts ("Actionati o tasta pentru a continua");
 getch();
}
```

2.12 Să se scrie un program care afişează constanta 12345 în zecimal, octal şi hexazecimal.

PROGRAMUL BII12

```
#include <stdio.h>
#include <conio.h>

#define V 12345

main() /* afiseaza intregul 12345 in zecimal, octal si hexazecimal */
{
 printf("zecimal %d\n", V);
 printf("octal %o\n", V);
 printf("hexazecimal %x\n", V);
 puts("Actionati o tasta pentru a continua");
 getch();
}
```

2.13 Să se scrie un program care afişează constanta 123456789 în zecimal, octal şi hexazecimal

PROGRAMUL BII13

```
#include <stdio.h>
#include <conio.h>
```

```
#define V 123456789

main() /* afiseaza constanta 123456789 in zecimal, octal
          si hexazecimal */
{
 printf("zecimal %ld\n", V);
 printf("octal %lo\n", V);
 printf("hexazecimal %lx\n", V);
 puts("Actionati o tasta pentru a continua");
 getch();
}
```

2.14 Programul BII14 afişează constanta 123 cu diferiţi specificatori de format.

PROGRAMUL BII14

```
#include <stdio.h>
#include <conio.h>
#define V 123

main() /* afiseaza constanta 123 cu diferiti specificatori de format */
{
 printf("*%d*\n", V);
 printf("*%7d*\n", V);
 printf("*%-7d*\n", V);
 printf("*%07d*\n", V);
 puts("Actionati o tasta pentru a continua");
 getch();
}
```

Rezultatele execuţiei programului sînt:

```
*123*
*    123*
*123    *
*0000123*
Actionati o tasta pentru a continua
```

2.15 Programul BII15 afişează numere neîntregi folosind diferiţi specificatori de format.

PROGRAMUL BII15

```
#include <stdio.h>
#include <conio.h>

#define A 47.389
#define X -123.5e20

main() /* afiseaza numere neintregi cu diferiti specificatori de format*/
{
 printf("A=%f\n", A);
 printf("A=%.3f\n", A);
 printf("A=%.2f\n", A);
 printf("A=%.1f\n", A);
 printf("A=%.0f\n", A);
 printf("X=%e\n", X);
 printf("X=%.4e\n", X);
 printf("X=%.3e\n", X);
 printf("X=%.2e\n", X);
 printf("X=%.1e\n", X);
 printf("A=%g X=%G\n", A, X);
 printf("%20f\n%20e\n%20g\n", 1.25,1.25,1.25);
 printf("%20f\n%20e\n%20g\n",.25,.25,.25);
 printf("%20f\n%20e\n%20g\n",123e-1,123e-1,123e-1);
 printf("%20f\n%20e\n%20g\n", 12.3e4,12.3e4,
         12.3e4);
 printf("%30.20Le\n",356.78214365789121389l);
 printf("%40.30Le\n",
        31415926535897932384526434e-25l);
}
```

Rezultatele execuţiei programului sînt:

```
A=47.389000
A=47.389
A=47.39
A=47.4
A=47
X=-1.235000e+22
X=-1.2350e+22
X=-1.235e+22
X=-1.24e+22
X=-1.2e+22
A=47.389 X=-1.235E+22
```

```
        1.250000
      1.250000e+00
            1.25
        0.250000
      2.500000e-01
            0.25
       12.300000
    1.230000e+01
            12.3
   123000.000000
    1.230000e+05
          123000
3.5678214365789121400e+02
3.14159265358979324000000000000000e+00
```

2.6. Funcţia scanf

Funcţia *scanf* poate fi folosită pentru a introduce date tastate la terminalul standard sub controlul unor formate. Ea poate fi apelată printr-o construcţie de forma:

scanf(*control,par1,par2,...,parn*);

unde:

control - Este un şir de caractere care defineşte formatele datelor şi a eventualelor texte aflate la intrarea de la tastatură.

par1,par2,...,parn - Sînt adresele zonelor în care se păstrează datele citite după ce au fost convertite din formatele lor externe în formate interne corespunzătoare.

Caracterele albe din compunerea parametrului *control* sînt neglijate. Restul caracterelor, care nu fac parte dintr-un specificator de format, trebuie să existe la intrare în poziţii corespunzătoare. Acestea se folosesc în vederea efectuării de controale asupra datelor citite.

De exemplu, dacă o dată care se citeşte se atribuie variabilei x, atunci data respectivă poate fi precedată de textul:

x=

În acest caz parametrul trebuie să conţină în poziţia corespunzătoare:

x=specificatorul de format al datei care se citeşte

Textele de la intrare pot fi precedate de caractere albe care se neglijează.

Specificatorii de format controlează introducerea datelor care au diferite formate externe reprezentate prin anumite succesiuni de caractere. De asemenea, specificatorii de format definesc conversiile din formate externe (ale datelor) în cele interne.

Specificatorii de format sînt asemănători cu cei întîlniţi la funcţia *printf*. Ei încep cu un caracter procent şi se termină cu 1-2 litere. Literele definesc tipul conversiei.

Între procent şi litere, într-un specificator de format mai putem utiliza, în ordine:

- un caracter asterisc opţional;
- un şir de cifre opţional, care defineşte lungimea maximă a cîmpului din care se citeşte data sub controlul formatului respectiv.

Cîmpul controlat de un specificator de format *începe* cu *primul* caracter curent care nu este alb şi se *termină*:

a. fie la caracterul după care urmează un caracter alb;

b. fie la caracterul după care urmează un caracter care nu corespunde specificatorului de format care controlează acel cîmp;

c. fie la caracterul prin care se ajunge la lungimea maximă a cîmpului, indicată în specificatorul de format.

Condiţia c. este absentă dacă în specificatorul de format nu este indicată lungimea maximă a cîmpului.

Menţionăm că această regulă nu se aplică la citirea de caractere.

În cazul în care specificatorul de format conţine caracterul asterisc, data din cîmpul respectiv va fi prezentă la intrare dar ea nu se atribuie nici unei variabile şi deci nu-i va corespunde un parametru, spre deosebire de ceilalţi specificatori care nu conţin caracterul asterisc.

Funcţia *scanf* citeşte toate cîmpurile care corespund specificatorilor de format, inclusiv eventualele texte prezente în parametrul *control*.

În cazul unei erori, citirea se întrerupe în locul în care s-a întîlnit eroarea. Eroarea poate proveni:

- din necorespondenţa textului curent din parametrul de control cu cel din fişierul de intrare;
- din neconcordanţa dintre data din cîmp şi specificatorul de format sub controlul căruia se face citirea.

Apariţia unei erori poate fi pusă uşor în evidenţă deoarece *scanf*

returnează, la revenirea din ea, numărul cîmpurilor citite corect. În felul acesta, valoarea returnată de *scanf* poate fi testată şi se poate stabili cîte cîmpuri au fost citite la fiecare apel a lui *scanf*.

Funcţia *scanf* citeşte date din zona tampon ataşată tastaturii, la fel ca şi *getchar*. De aceea, datele se citesc efectiv după ce s-a acţionat tasta *Enter*.

La întîlnirea sfîrşitului de fişier (<CTRL>-Z la mediile Turbo C şi Turbo C++) se returnează valoarea *EOF*. Se recomandă ca sfîrşitul de fişier să fie precedat de un caracter alb pentru a nu pierde data tastată înainte de sfîrşitul de fişier. De asemenea, după sfîrşitul de fişier se acţionează tasta Enter pentru a termina înregistrarea curentă.

Funcţia *scanf*, ca şi funcţia *printf* are prototipul în fişierul *stdio.h*.

Aşa cum s-a arătat mai sus, parametrii *par1, par2, ..., parn* sînt adresele zonelor de memorie în care se păstrează datele citite de la tastatură în urma conversiilor în format intern. Adresa unei zone de memorie se exprimă adesea folosind *operatorul unar &*.

Aşa de exemplu, dacă *i* este o variabilă simplă oarecare, atunci

&i

reprezintă adresa zonei de memorie alocată variabilei i.

În general, dacă *nume* este numele unei variabile simple, atunci adresa de început a zonei de memorie alocată variabilei nume se exprimă cu ajutorul expresiei:

&nume

Amintim că dacă *nume* este numele unui tablou, atunci nu vom mai folosi operatorul &, deoarece în acest caz nume are ca valoare chiar adresa de început a zonei de memorie alocată tabloului respectiv.

2.6.1. Litera c

Specificatorul de format

%c

se foloseşte pentru a citi, cu ajutorul lui *scanf*, caracterul curent din zona tampon corespunzătoare datelor introduse de la tastatură.

În acest caz nu se face avans pînă la primul caracter care nu este alb, ci pur şi simplu se citeşte caracterul curent (unul singur). Codul său ASCII se păstrează în zona de memorie definită de parametrul efectiv corespunzător.

Exemplu:

Fie declaraţia:

```
char car;
```

Apelul:

```
scanf("%c",&car);
```

citeşte caracterul curent din zona tampon şi păstrează codul ASCII al acestuia în zona de memorie alocată variabilei car.

Apelînd în continuare funcţia *printf*:

```
printf("%c",car);
```

se va afişa, la terminalul standard, caracterul citit prin apelul de mai sus a lui *scanf*.

2.6.2. Litera s

Specificatorii de format terminaţi prin *s* se folosesc pentru a citi şiruri de caractere.

La citirea cu un astfel de specificator de format, data se consideră ca făcînd parte dintr-un cîmp care începe cu un caracter care nu este alb şi care se termină fie la caracterul după care urmează un caracter alb, fie la caracterul prin care se ajunge la lungimea maximă de caractere indicată în specificatorul de format.

Caracterele citite se pot păstra într-o zonă de memorie organizată ca un tablou de caractere. În acest caz parametrul corespunzător specificatorului de format poate fi chiar numele acestui tablou.

Menţionăm că după ultimul caracter citit sub controlul unui astfel de specificator de format se păstrează în mod automat caracterul NUL ('\0').

Exemple:

1. Fie textul

   Limbajul C++

   tastat pe un acelaşi rînd la terminalul standard.

   Dacă *tab1* şi *tab2* sînt declarate ca mai jos:

   ```
   char tab1[10];
   char tab2[4];
   ```

   atunci textul de mai sus poate fi citit folosind apelurile:

```
scanf("%s",tab1); //se citeste textul: Limbajul
                  //cimpul incepe cu litera L si se termina cu l
                  //deoarece dupa l urmeaza spatiu

scanf("%s",tab2); //se citeste textul: C++
                  //cimpul incepe cu litera C si se termina cu cel
                  //de al doilea caracter + deoarece dupa el
                  //urmeaza newline
```

Acelaşi lucru se poate realiza folosind un singur apel a lui *scanf* cu doi specificatori de format:

scanf("%s %s",tab1,tab2);

Menţionăm că spaţiul dintre specificatorii de format nu este esenţial, el se neglijează. Deci acelaşi apel poate fi scris şi sub forma

scanf("%s%s",tab1,tab2);

Datele citite prin aceste apeluri se păstrează în memorie prin codurile lor ASCII astfel:

```
tab1[0]->'L', tab1[1]->'i', tab1[2]->'m',
tab1[3]->'b', tab1[4]->'a', tab1[5]->'j',
tab1[6]->'u', tab1[7]->'l', tab1[8]->'\0',
tab2[0]->'C', tab2[1]->'+', tab2[2]->'+',
tab2[3]->'\0'.
```

2. Fie declaraţiile:

char tab1[10];

char tab2[10];

şi apelul:

scanf("%2s%9s",tab1,tab2);

Presupunem că la terminal s-a tastat cuvîntul *necunoscut* precedat şi urmat de cel puţin un caracter alb.

În acest caz funcţia *scanf* realizează următoarele:

- Primul specificator de format asigură citirea cuvîntului:

  ne

  Aceasta deoarece cîmpul are lungimea maximă de 2 caractere. El începe cu litera "n" (primul caracter care nu este alb) şi se termină cu litera "e" la care s-a ajuns să se citească două caractere, cît s-a specificat în specificatorul de format;

- Al doilea specificator de format asigură citirea în continuare. În acest caz cîmpul începe cu litera "c" şi se termină cu litera "t" deoarece după *t* urmează un caracter alb, iar numărul total al literelor cuvîntului *cunoscut* este 8, care este mai mic decît lungimea maximă admisă: 9.

În concluzie, după apelul de mai sus, vom avea:

tab1[0]->'n'
tab1[1]->'e'
tab1[2]->'\0'
tab2[0]->'c'
tab2[1]->'u'
...
tab2[7]->'t'
tab2[8]->'\0'

3. Specificatorul

 %1s

 permite citirea unui singur caracter. O primă deosebire dintre acesta şi specificatorul %c constă în aceea că, în timp ce %1s citeşte primul caracter care nu este alb, specificatorul %c citeşte caracterul curent, indiferent dacă acesta este alb sau nu. Deci cei doi specificatori citesc acelaşi lucru numai dacă, caracterul curent nu este alb.

O altă deosebire dintre cei doi specificatori constă în aceea că, specificatorul %1s implică memorarea caracterului *NUL* după caracterul citit, ceea ce nu are loc în cazul specificatorului %c. De aceea, secvenţa:

```
char car;
...
scanf("%c",&car);
```

se modifică dacă folosim specificatorul %1s, astfel:

```
char car[2];
...
scanf("%1s",car);
```

2.6.3. Litera d

Specificatorii terminaţi prin litera *d* permit citirea întregilor zecimali şi conversia lor spre tipul *int*.

Numărul poate fi precedat de un semn. El trebuie să aparţină intervalului [-32768,32767]. În caz contrar, rezultatul va fi imprevizibil.

Cîmpul din care se citeşte întregul se defineşte conform regulei generale indicate mai sus.

Exemple:

1. Fie declaraţia

```
int n;
```

şi apelul

```
scanf("%d",&n);
```

Dacă în cîmpul curent din zona tampon de intrare se află întregul

123

precedat şi urmat de cel puţin un caracter alb, atunci după acest apel se va păstra, în zona de memorie alocată lui n, valoarea în binar a întregului zecimal 123.

La revenire funcţia scanf returnează valoarea 1 (un cîmp citit corect).

2. Fie declaraţiile

```
int i1,i2,i3;
```

şi apelul:

```
scanf("%2d %3d %2d",&i1,&i2,&i3);
```

Dacă în zona de intrare se află succesiunea de cifre

1234567

precedată şi urmată de caractere albe, atunci după apel variabilele i1, i2 şi i3 au respectiv valorile 12, 345 şi 67.

Într-adevăr, primul specificator de format asigură citirea cifrelor dintr-un cîmp de 2 cifre, al doilea specificator dintr- un cîmp de 3 cifre, iar al treilea dintr-un cîmp de 2 cifre. La revenire funcţia *scanf* returnează valoarea 3 (3 cîmpuri citite corect).

3. Fie declaraţia

```
int n;
```

şi apelul

```
scanf("%d",&n);
```

Dacă, la intrare, construcţia curentă este

i23

atunci construcţia respectivă se consideră eronată, deoarece ea nu este un întreg zecimal. În acest caz *scanf* nu citeşte nimic şi se revine din ea cu valoarea zero. Construcţia respectivă va rămîne curentă, deci ea poate fi citită apelînd în continuare funcţia *scanf* cu un alt specificator de format (de exemplu %s) sau macroul *getchar*.

4. Considerăm declaraţia şi apelul lui *scanf* din exemplul precedent.

   Dacă la intrare se află construcţia

   23i

   atunci se va citi întregul 23.

Cîmpul începe cu cifra 2 şi se sfîrşeşte cu cifra 3, deoarece după ea urmează o literă, adică un caracter care nu intră în compunerea întregilor zecimali.

La un nou apel al funcţiei *scanf* se va încerca citirea literei *i*.

După citirea întregului 23 se va reveni din *scanf* cu valoarea 1.

2.6.4. Litera o

Litera *o* se utilizează ca şi litera *d*, deosebirea constă în aceea că în loc să se citească un întreg zecimal, se va citi un întreg *octal*. În acest caz fiecare cifră octală se înlocuieşte cu trei cifre binare care definesc aceeaşi valoare ca şi cifra respectivă.

2.6.5. Literele x sau X

Ambele litere se utilizează la fel ca şi litera *d*, deosebirea constă în aceea că în loc să se citească un întreg zecimal, se va citi un întreg *hexazecimal*. În acest caz fiecare cifră hexazecimală se înlocuieşte cu 4 cifre binare care definesc aceeaşi valoare ca şi cifra respectivă.

Cifrele hexazecimale mai mari decît 9 pot fi reprezentate prin litere mici sau mari.

2.6.6. Litera u

Se utilizează pentru a citi întregi zecimali care se păstrează convertiţi în binar şi au tipul *unsigned*.

2.6.7. Litera f

Litera *f* permite citirea numerelor zecimale care pot fi reprezentate atît cu exponent, cît şi fără exponent şi păstrarea lor în format *flotant simplă precizie* deci date de tip *float*.

2.6.8. Litera l

Litera *l* poate precede pe oricare din literele *d, o, x, X, u* sau *f*.

Dacă *l* precede pe una din literele *d, o, x* sau *X*, atunci data citită se va converti spre tipul *long* în loc de *int*.

Specificatorul terminat cu *lu* realizează conversia spre *unsigned long*.

Specificatorul *lf* realizează conversia numărului citit în *flotantă dublă precizie*, deci spre tipul *double*.

2.6.9. Litera L

Litera *L* se foloseşte la scrierea specificatorilor terminaţi prin *Lf*. Ea permite conversia numerelor zecimale atît cu exponent, cît şi fără exponent, în format *flotant* de tip *long double*.

Exerciţii:

2.16 Să se scrie un program care citeşte un caracter de la terminalul standard cu ajutorul funcţiei *scanf* şi afişează codul ASCII al caracterului respectiv.

PROGRAMUL BII16

```
#include <stdio.h>
#include <conio.h>

main() /* - citeste un caracter folosind functia scanf;
          - afiseaza codul ASCII al caracterului respectiv. */
{
 char car;

 scanf("%c", &car);
 printf("%d\n",car);
 puts("Actionati o tasta pentru a continua");
 getch();
}
```

2.17 Să se scrie un program care citeşte numele şi prenumele unei persoane separate prin spaţii albe, apoi afişează prenumele pe un rînd şi numele pe rîndul următor.

PROGRAMUL BII17

```
#include <stdio.h>
#include <conio.h>

main() /* - citeste numele si prenumele unei persoane separate
            prin spatii albe;
          - afiseaza prenumele pe un rind si numele pe
            rindul urmator */
{
 char nume[255];
 char prenume[255];

 scanf("%s %s", nume, prenume);
 puts(prenume);
 puts(nume);
 puts("Actionati o tasta pentru a continua");
 getch();
}
```

2.18 Să se scrie un program care citeşte un întreg zecimal de cel mult 4 cifre şi-l afişează în zecimal, octal şi hexazecimal.

PROGRAMUL BII18

```
#include <stdio.h>
main() /* citeste un intreg zecimal de cel mult 4 cifre si-l
          afiseaza in zecimal, octal si hexazecimal */

{
 int i;

 scanf("%d", &i);
 printf("%d\n", i);
 printf("%o\n", i);
 printf("%x\n", i);
}
```

Observaţie:

La execuţia acestui program nu se mai afişează fereastra utilizator cu rezultatele. Pentru vizualizarea ei se tastează

<Alt>-F5

Se revine în fereastra de editare acţionînd o tastă oarecare.

2.19 Să se scrie un program care citeşte un întreg zecimal de cel mult 9 cifre şi-l afişează în zecimal, octal şi hexazecimal.

PROGRAMUL BII19

```
#include <stdio.h>
main() /* citeste un intreg zecimal de cel mult 9 cifre si-l
             afiseaza in zecimal, octal si hexazecimal */
{
 long i;

 scanf("%ld", &i);
 printf("%ld\n%lo\n%lx\n",i,i,i);
}
```

2.20 Să se scrie un program care:

- citeşte o dată calendaristică tastată sub forma:
  zzllaa;
- afişează data respectivă sub forma:
  19aa/ll/zz

unde:

zz	- Două cifre ce reprezintă ziua din lună: 01, 02, ...
ll	- Două cifre ce reprezintă numărul lunii: 01, 02, ..., 12.
aa	- Două cifre ce reprezintă ultimele două cifre ale unui an calendaristic care începe cu 19.

Programul are la intrare o succesiune de 6 cifre care pot fi precedate şi urmate de spaţii albe. Ele pot fi citite cîte două folosind de trei ori specificatorul de format %2d.

PROGRAMUL BII20

```
#include <stdio.h>
```

```
main() /* - citeste o data calendaristica sub forma: zzllaa;
          - o afiseaza sub forma: 19aa/ll/zz. */
{
 int zi;
 int luna;
 int an;

 scanf("%2d %2d %2d", &zi,&luna,&an);
 printf("19%02d/%02d/%02d\n",an,luna,zi);
}
```

Observaţie:

La afişare s-a folosit specificatorul de format:

%02d

Acesta afişează cifrele cu un zero nesemnificativ în faţă. Deci cifrele 1,2,...,9 se afişează ca 01,02,...,09.

2.21 Să se scrie un program care citeşte datele de mai jos şi apoi afişează tipul operaţiei citite pe un rînd, iar celelalte date pe rîndul următor:

Tip operaţie	denumire	UM	cod	preţ	cantitate
I	REZISTENTA	010KO	123456789	15	1000

PROGRAMUL BII21

```
#include <stdio.h>

#define MAXDEN 30

main() /* citeste un rind de date de la intrarea standard si le
          afiseaza astfel:
            - tip operatie pe un rind;
            - celelalte date pe rindul urmator. */
{
 char den[MAXDEN+1];
 char tip[2];
 int val;
 char unit[3];
 long cod;
```

```
float pret,cantitate;

scanf("%1s%30s %3d %2s %ld %f %f",tip,den,&val,
     unit,&cod,&pret,&cantitate);
printf("%s\n",tip);
 printf("%s %03d%s %ld %f %f\n",den,val,unit,
        cod,pret,cantitate);
}
```

Observaţie:

Tipul este un caracter:

- I (intrare);
- E (ieşire);
- T (transfer) etc.

Citirea lui se face folosind specificatorul %1s. Acesta permite eliminarea eventualelor caractere albe care ar putea precede litera ce defineşte tipul.

3. EXPRESII

O *expresie*, în limbajul C, este formată dintr-un *operand* sau mai mulţi legaţi prin *operatori*.

3.1. Operand

Un operand poate fi:

- o constantă;
- o constantă simbolică;
- numele unei variabile simple;
- numele unui tablou;
- numele unei structuri;
- numele unui tip;
- numele unei funcţii;
- referirea la elementul unui tablou (variabilă cu indici);
- referirea la elementul unei structuri;
- apelul unei funcţii;
- expresie inclusă în paranteze rotunde.

Unui operand îi corespunde un *tip* şi o *valoare*. Dacă tipul operandului este bine precizat la compilare, valoarea operandului se determină fie la compilare, fie la execuţie.

Exemple:

1. 1234

Este o constantă întreagă zecimală de tip *int*. Reprezintă un operand constant de tip *int*.

2. 0xa1b2

Este o constantă întreagă hexazecimală de tip *unsigned*. Reprezintă un operand constant de tip *unsigned*.

3. Fie declaraţia

float a1b2;

Numele

a1b2

este numele unei variabile simple, deci reprezintă un operand. El are tipul *float*.

4. Fie declaraţia

```
int tab[10];
tab[2]
```

Este o referire la elementul al treilea al tabloului *tab*. Reprezintă un operand de tip *int*.

Numele

tab

este numele unui tablou şi deci el reprezintă un operand. El este de tip "adresă de întregi de tip int" (*pointer to int*), adică are ca valoare adresa unei zone de memorie care conţine întregi de tip *int*. Se utilizează în mod frecvent ca parametru în apelul funcţiilor.

5. sum(n,x)

Este un apel al funcţiei *sum*. Reprezintă un operand al cărui tip coincide cu tipul valorii returnate de funcţia *sum*. Funcţia *sum* poate fi apelată ca operand dacă returnează o valoare.

6. (expresie)

Este un operand al cărui tip coincide cu tipul expresiei inclusă între paranteze.

3.2. Operatori

Operatorii pot fi *unari* sau *binari*.

Un operator unar se aplică unui singur operand.

Un operator binar se aplică la doi operanzi. Operatorul binar se aplică la operandul care îl precede imediat şi la care îl urmează imediat.

Operatorii limbajului C nu pot avea ca operanzi constante şir (şiruri de caractere). De asemenea, există limite în aplicarea operatorilor la anumiţi

operanzi.

Mai jos, se descriu operatorii limbajului C. Menţionăm că limbajul C++ conţine şi alţi operatori. Ei vor fi descrişi ulterior, pe măsură ce se vor defini construcţiile specifice limbajului C++.

Operatorii limbajului C pot fi grupaţi în mai multe clase: operatori aritmetici, operatori de relaţie, operatori logici etc.

La scrierea unei expresii se pot folosi operatori din aceeaşi clasă sau din clase diferite. În principiu, se pot defini expresii complexe în care să se utilizeze operatori din toate clasele. La evaluarea unei astfel de expresii este necesar să se ţină seama de *priorităţile* operatorilor care aparţin diferitelor clase de operatori, de *asociativitatea* operatorilor de aceeaşi prioritate şi de *regula conversiilor implicite*.

3.2.1. Operatorii aritmetici

Operatorii aritmetici se utilizează la efectuarea calculelor cu date de diferite tipuri predefinite. Aceştia sînt:

a. operatorii unari + şi -;

b. operatorii binari multiplicativi * / şi %;

c. operatorii binari aditivi + şi -.

Operatorii unari sînt mai prioritari decît cei binari, iar operatorii multiplicativi sînt mai prioritari decît cei aditivi.

Operatorii unari au aceeaşi prioritate. Ei se asociază de la dreapta spre stînga.

Operatorii multiplicativi au aceeaşi prioritate. De asemenea, operatorii aditivi au aceeaşi prioritate.

Operatorii binari se asociază de la stînga la dreapta.

Operatorul unar + nu are nici un efect.

Operatorul unar - are ca efect negativarea valorii operandului pe care-l precede.

Operatorul * reprezintă operatorul de inmulţire al operanzilor la care se aplică.

Operatorul / reprezintă operatorul de împărţire a valorii operandului care îl precede la valoarea operandului care îl urmează.

În cazul în care ambii operanzi sînt întregi (*int, unsigned* sau *long*), se

realizează o împărţire întreagă, adică rezultatul împărţirii este partea întreagă a cîtului.

Operatorul % se aplică numai la operanzi întregi şi are ca rezultat restul împărţirii întregi a valorilor operanzilor săi.

Operatorii binari + şi - reprezintă operaţiile obişnuite de adunare şi scădere definite în matematică.

Exemple:

1. 1234 şi +1234

   sînt expresii care au aceeaşi valoare şi tip. Prima se reduce la operandul 1234 care este o constantă de tip *int*.

A doua se compune din operandul 1234 la care se aplică operatorul unar +. Cum acesta nu are nici un efect, cele două expresii au aceeaşi valoare şi tip.

2. -1234

   este o expresie care se compune din operandul 1234 la care se aplică operatorul unar de negativare.

3. 7/3

Deoarece operanzii sînt întregi, se realizează împărţirea întreagă a lui 7 la 3. Se obţine valoarea 2 de tip *int*.

4. 7%3

   Expresia are ca valoare restul împărţirii lui 7 la 3, deci 7%3=1.

5. -7%3

   Expresia are ca valoare restul împărţirii lui -7 la 3, deci -7%3=-1.

   Într-adevăr,

   -7/3=-2 şi -7=(-2)*3+(-1),

deci restul împărţirii este -1.

Exerciţii:

3.1 Să se scrie un program care citeşte valoarea lui *x*, calculează valoarea

polinomul p(x)=3x**2-8x+7 şi afişează rezultatul. Variabila *x* este de tip întreg.

Menţionăm că aici şi în continuare prin a**b notăm *a* la puterea *b*.

PROGRAMUL BIII1

```
#include <stdio.h>
main() /* - citeste valoarea lui x;
            - calculeaza valoarea polinomului 3x**2 - 8x + 7;
            - afiseaza rezultatul. */
{
 int x;

 printf("tastati valoarea lui x=");
 scanf("%d", &x);
 printf("x=%d\tp(x)=%d\n", x, 3*x*x - 8*x + 7 );
}
```

Observaţie:

Rezultatul calculului este incorect dacă cel puţin un rezultat parţial depăşeşte intervalul valorilor de tip *int* ([-32768,32767]).

În general, dacă operanzii unui operator binar au acelaşi tip *t*, atunci rezultatul aplicării operatorului are tipul *t*. În cazul în care rezultatul este în afara intervalului de valori corespunzător tipului *t*, rezultatul aplicării operatorului este eronat.

Dacă un operator binar se aplică la operanzi de tipuri diferite, se aplică regula conversiilor implicite (vezi mai jos).

3.2 Să se scrie un program care citeşte valoarea lui *x*, calculează valoarea polinomului f(x)=3,5x**3-9,8x+3,7 şi afişează rezultatul. Calculele se fac în flotantă dublă precizie.

PROGRAMUL BIII2

```
#include <stdio.h>
main() /* - citeste valoarea lui x;
            - calculeaza si afiseaza valoarea polinomului
                  3,5x**3 - 9,8x + 3,7. */
{
 double x;
```

```
 printf("tastati valoarea lui x = ");
 scanf("%lf", &x);
 printf("x=%g\tf(x)=%g\n",x, 3.5*x*x*x -
        9.8*x + 3.7);
}
```

Observaţie:

Constantele flotante 3.5, 9.8 şi 3.7 se consideră de tip *double*.

3.3 Să se scrie un program care citeşte valoarea lui x, calculează valoarea polinomului a(x)=3x**20-6x**16+8X**9-7x**5+1 şi afişează rezultatul. Calculele se fac în flotantă dublă precizie.

În acest exemplu intervin puteri mari ale lui *x*. De aceea, pentru a evita expresii de forma:

x*x*...*x

unde *x* să se repete de 20 de ori, vom apela funcţia *pow* care realizează ridicarea la putere. Ea are prototipul definit în fişierul *math.h* şi acesta este:

double pow(double *x*, **double** *y*);

La revenire se returnează x**y, rezultatul fiind de tip *double*.

PROGRAMUL BIII3

```
#include <stdio.h>
#include <math.h>

main() /* - citeste valoarea lui x;
          - calculeaza si afiseaza valoarea polinomului:
                a(x)=3x**20-6x**16+8x**9-7x**5+1. */
{
 double x;

 printf("tastati valoarea lui x = ");
 scanf("%lf",&x);
 printf("x=%g\ta(x)=%g\n",x,
        3.0*pow(x,20.0) - 6.0*pow(x,16.0) +
        8.0*pow(x,9.0) - 7.0*pow(x,5.0)
         + 1.0);
}
```

3.4 Să se scrie un program care citeşte valoarea variabilei *x* şi a

coeficienţilor polinomului:

q(x)=c4*x**4+c3*x**3+c2*x**2+c1*x+c0,

calculează valoarea polinomului q(x) şi afişează rezultatul. Calculele se fac în flotantă dublă precizie.

Calculul valorii unui polinom se poate face folosind funcţia de bibliotecă *poly*. Ea are prototipul definit în fişierul *math.h* şi este:

double poly(double *x*, **int** *n*, **double** *c[]*);

Returnează valoarea polinomului de grad *n*, pentru valoarea lui *x*, coeficienţii polinomului fiind c[0],c[1],...c[n]; c[0] este termenul liber; c[1] este coeficientul lui *x*; în general, c[i] este coeficientul lui *x* la puterea *i*.

PROGRAMUL BIII4

```
#include <stdio.h>
#include <math.h>

main() /* - citeste valoarea lui x si coeficientii c[0],c[1],...,c[4] ai
            polinomului
              q(x)=c[4]*x**4+c[3]*x**3+c[2]*x**2+c[1]*x+c[0]
          - calculeaza valoarea polinomului q(x);
          - afiseaza rezultatul. */
{
 double x,c[5];

 printf("valoarea lui x = ");
 scanf("%lf",&x);
 printf("coeficientii polinomului\n");
 printf("c0 = ");
 scanf("%lf",&c[0]);
 printf("c1 = ");
 scanf("%lf",&c[1]);
 printf("c2 = ");
 scanf("%lf",&c[2]);
 printf("c3 = ");
 scanf("%lf",&c[3]);
 printf("c4 = ");
 scanf("%lf",&c[4]);
 printf("x=%g\tq(x)=%g\n",x,poly(x,4,c));
}
```

3.5 Să se scrie un program care citeşte cu ecou de la tastatură o literă mică

şi o afişează ca literă mare.

Conversiile literelor mici în litere mari se realizează simplu în cazul utilizării codului ASCII. Într-adevăr, codurile ASCII ale literelor mici se află în intervalul [97,122], iar ale literelor mari în intervalul [65,90]. Aceste coduri corespund literelor în ordine alfabetică:

Litera mică	**Codul ASCII**	**Litera mare**	**Codul ASCII**
a	97	A	65
b	98	B	66
c	99	C	67
,..			
z	122	Z	90

Codul ASCII al unei litere mari se obţine din codul ASCII al aceleeaşi litere mici, scăzînd valoarea 32. Deci, dacă variabila *lit* are ca valoare codul ASCII al unei litere mici, atunci expresia:

lit-32

are ca valoare codul ASCII al aceleeaşi litere, dar mari.

De obicei, diferenţa 32 dintre codurile ASCII ale aceleeaşi litere, se exprimă mai sugestiv prin expresia:

'a'-'A'

Această expresie are doi operanzi care sînt constante caracter, deci fiecare are tipul *int*. Înseamnă că tipul expresiei de mai sus este *int*. În acest fel, codul ASCII al literei mari se obţine cu ajutorul expresiei:

lit-('a'-'A')

care se scrie mai simplu:

lit-'a'+'A'

PROGRAMUL BIII5

```
#include <conio.h>
main() /* citeste cu ecou de la tastatura o litera mica si o afiseaza
          ca litera mare */
{
 putch(getche() - 'a' + 'A');
}
```

Observaţie:

În acest program se presupune că se tastează la terminal o literă mică.

3.6 Să se scrie un program care citeşte cu ecou de la tastatură o literă mare şi o afişează ca literă mică.

În acest caz codul literei mari se adună cu valoarea 32, adică cu diferenţa 'a'-'A'.

PROGRAMUL BIII6

```
#include <conio.h>
main() /* citeste cu ecou de la tastatura o litera mare si o afiseaza
          ca litera mica */
{
 putch(getche() + 'a' - 'A');
}
```

3.7 Să se scrie un program care citeşte un număr ce reprezintă aria unei suprafeţe exprimată în jugăre.

Se cere să se exprime aria respectivă în hectare, prăjini pătrate şi stînjeni pătraţi.

Avem:

1 jugăr = 576 prăjini pătrate = 1600 stînjeni pătraţi = 5754,6415 metri pătraţi.

PROGRAMUL BIII7

```
#include <stdio.h>
main() /* - citeste un numar ce exprima aria unei suprafete in jugare;
          - afiseaza aria respectiva in:
          - prajini patrate;
          - stinjeni patrati;
          - hectare. */
{
 double aria;

 scanf("%lf", &aria);
 printf("aria=%g jugare\n", aria);
 printf("aria=%g prajini patrate\n",aria*576);
 printf("aria=%g stinjeni patrati\n",aria*1600);
```

```
    printf("aria=%g hectare\n",aria*0.57546415);
}
```

3.2.2. Regula conversiilor implicite

Această regulă se aplică la evaluarea expresiilor. Ea acţionează atunci cînd un operator binar se aplică la doi operanzi de tipuri diferite.

În cazul în care un operator se aplică la doi operanzi de tipuri diferite, operandul de tip "inferior" se converteşte spre tipul "superior" al celuilalt operand şi rezultatul este de tipul "superior".

Regula conversiilor implicite are în vedere şi nişte conversii generale independente de operatori. Ea se aplică în mai mulţi paşi, în ordinea indicată mai jos.

Înainte de toate se convertesc operanzii de tip *char* şi *enum* (acest tip va fi definit mai tîrziu) în tipul *int*.

Dacă operatorul curent se aplică la operanzi de acelaşi tip, atunci se execută operatorul respectiv, iar tipul rezultatului coincide cu tipul comun al operanzilor. Dacă rezultatul aplicării operatorului reprezintă o valoare în afara limitelor tipului respectiv, atunci rezultatul este eronat (are loc o "depăşire").

Dacă operatorul binar curent se aplică la operanzi diferiţi, atunci se face o conversie înainte de execuţia operatorului, conform paşilor de mai jos:

1. Dacă unul din operanzi este de tip *long double*, atunci celălalt operand se converteşte spre tipul *long double* şi rezultatul aplicării operatorului este de tip *long double*.

2. Altfel, dacă unul din operanzi este de tip *double*, atunci celălalt operand se converteşte spre tipul *double* şi rezultatul aplicării operatorului este de tip *double*.

3. Altfel, dacă unul din operanzi este de tip *float*, atunci celălalt operand se converteşte spre tipul *float* şi rezultatul aplicării operatorului este de tip *float*.

4. Altfel, dacă unul din operanzi este de tip *unsigned long*, atunci celălalt operand se converteşte spre *unsigned long* şi rezultatul aplicării operatorului este de tip *unsigned long*.

5. Altfel, dacă unul din operanzi este de tip *long*, atunci celălalt operand se converteşte spre tipul *long* şi rezultatul aplicării operatorului este de

tip *long*.

6. Altfel, unul din operanzi trebuie să fie de tip *unsigned*, celălalt de tip *int* şi acesta se converteşte spre *unsigned*, iar rezultatul aplicării operatorului este de tip *unsigned*.

Aplicînd regula de mai sus, la fiecare operator curent, în procesul de evaluare a unei expresii, se determină în final tipul expresiei respective.

Exemple:

1. int c;

   ...

   c-'a'+'A'

Constantele caracter 'a' şi 'A' sînt de tip *int*, deci toţi operanzii sînt de tip *int*. Nu este necesară nici o conversie pentru evaluarea expresiei.

2. int i;

   ...

   12345*i

Operanzii sînt de tip *int*, expresia este de tip *int*; rezultatul înmulţirii este eronat dacă este în afara intervalului [-32768,32767].

3. int i;

   long double a;

   ...

   a*3+i

Ţinînd seama de prioritatea operatorilor, întîi se execută înmulţirea, apoi adunarea; expresia se evaluează astfel:

- 3 se converteşte spre *long double* (regula 1), se realizează înmulţirea şi rezultatul este de tip *long double*;
- la rezultatul înmulţirii se adună *i* după ce, în prealabil, *i* se converteşte spre *long double*, deci expresia este de tip *long double*.

Exerciţii:

3.8 Să se scrie un program care citeşte doi întregi de la intrarea standard şi afişează media lor cu o zecimală.

PROGRAMUL BIII8

```
#include <stdio.h>
main() /* citeste doi intregi si afiseaza media lor aritmetica
          cu o zecimala */
{
 int a,b;

 scanf("%d %d", &a,&b);
 printf("a=%d\tb=%d\tmedia=%.1f\n",a,b,(a+b)/2.0f);
}
```

Observaţii:

1. Pentru calculul mediei se foloseşte expresia:

   (a+b)/2.0f

   Ea se evaluează astfel:

   - Se calculează suma a+b; rezultatul este de tip *int*, deoarece ambii operanzi sînt de tip *int*. Dacă suma nu aparţine intervalului [-32768,32767], rezultatul este eronat.
   - Rezultatul adunării se converteşte spre *float*, deoarece celălalt operand este de tip *float*, apoi se face împărţirea şi rezultatul este de tip *float*.

2. Dacă s-ar fi utilizat expresia

   (a+b)/2.0

suma a+b s-ar fi convertit spre *double* deoarece împărţitorul este de tip *double*; în acest caz rezultatul este de tip *double*.

3. Expresia

   (a+b)/2

realizează împărţirea întreagă dintre operanzii a+b şi 2 care sînt de tip *int*; de aceea, pentru *a* şi *b* de parităţi diferite, expresia de mai sus nu dă rezultatul corect.

3.9 Să se scrie un program care citeşte măsura unui unghi în grade sexagesimale şi afişează valoarea funcţiei sinus pentru unghiul respectiv.

Măsura unghiului în grade sexagesimale se tastează sub forma:

g m s

unde:

g - Întreg ce reprezintă gradele.
m - Întreg ce reprezintă minutele.
s - Întreg ce reprezintă secundele.

Pentru calculul fucţiei sinus se apelează funcţia *sin* aflată în biblioteca limbajelor C şi C++.

Prototipul ei este:

double sin(double);

şi se află în fişierul *math.h*.

Parametrul funcţiei este măsura unghiului în radiani pentru care se calculează valoarea funcţiei *sin*. Conversia măsurii unghiului din grade sexagesimale în radiani se realizează folosind expresia:

(g+m/60.0+s/3600.0)*PI/180.0

unde

PI = 3.14159265358979

PROGRAMUL BIII9

```
#include <stdio.h>
#include <math.h>

#define PI 3.14159265358979

main() /* citeste masura unui unghi in grade sexagesimale si afiseaza
          valoarea functiei sin pentru unghiul respectiv. */
{
 int g,m,s;

 scanf("%d %d %d", &g,&m,&s);
 printf("grade:%d\tminute:%d\tsecunde:%d\n",g,m,s);
 printf("sin=%.13f\n",sin((g+m/60.0 +
 s/3600.0)*PI/180.0));
}
```

3.2.3. Operatorii de relaţie

Operatorii de relaţie sînt 4:

< - mai mic;
<= - mai mic sau egal;
> - mai mare;
>= - mai mare sau egal.

Ei au aceeaşi prioritate, care este mai mică decît a operatorilor aditivi.

O expresie de forma

E1 operator_de_relatie E2

unde:

E1 şi E2 - Sînt expresii, are una din valorile 0 sau 1.

Astfel, expresia are valoarea 1, dacă valorile celor două expresii satisfac relaţia prin care acestea sînt legate şi 0 în caz contrar.

De obicei, se spune despre o expresie relaţională că ea are valoarea *adevărat* dacă subexpresiile ei satisfac operatorul relaţional prin care sînt legate; în caz contrar se spune că expresia are valoarea *fals*.

În limbajele C şi C++ nu există valori logice speciale. Valoarea *fals* se reprezintă prin valoarea 0 (zero), iar valoarea *adevărat* printr-o valoare diferită de zero.

Exerciţii:

3.10 Să se scrie un program care citeşte două numere, primul este de tip *int*, iar al doilea este de tip *double*. Programul afişează valoarea 2 dacă numărul flotant este mai mare decît cel întreg şi 1 în caz contrar.

Presupunem că primul număr se atribuie variabilei *i*, iar cel de al doilea variabilei *f*.

Atunci expresia

f > i

are valoarea 1 dacă numărul flotant este mai mare decît cel întreg şi 0 în caz contrar. Pentru evaluarea relaţiei de mai sus, valoarea variabilei *i* se converteşte spre *double* şi abea apoi se execută operatorul de relaţie. Tipul expresiei este *int*.

Programul de faţă trebuie să afişeze valoarea acestei expresii mărite cu 1, ceea ce se obţine folosind expresia

(f > i)+1

Parantezele sînt necesare deoarece operatorul + este mai prioritar decît > . În lipsa lor, se compară valoarea lui *f* cu valoarea sumei i+1.

PROGRAMUL BIII10

```
#include <stdio.h>
main() /* - citeste pe i si f;
          - afiseaza 2 daca f este mai mare decit i si 1 in caz contrar */
{
 int i;
 double f;

 scanf("%d %lf", &i,&f);
 printf("i=%d\tf=%f\t%d\n",i,f,(f > i) + 1 );
}
```

3.2.4. Operatorii de egalitate

Operatorii de egalitate sînt doi:

== - egal;
!= - diferit.

Ei au aceeaşi prioritate care este imediat mai mică decît cea a operatorilor de relaţie.

O expresie de forma:

E1 operator_de_egalitate E2

unde:

E1 şi E2 - Sînt expresii, are una din valorile 0 sau 1.

Astfel, expresia de mai sus are valoarea 1, dacă valorile expresiilor E1 şi E2 satisfac operatorul de egalitate prin care sînt legate şi zero în caz contrar. În primul caz, se obişnuieşte să se spună că expresia de mai sus este *adevărată*, iar în cel de al doilea caz, că ea este *falsă*.

Exemple:

1. int a;

   long b;

   Expresia

   b==a*a

are valoarea adevărat (1) dacă *b* este egal cu pătratul lui *a*; altfel ea are valoarea fals (0).

2. int x;

Expresia

x%4==0

are valoarea adevărat dacă *x* este multiplu de 4 şi fals în caz contrar.

3. int x;

Expresia

x%100!=0

are valoarea adevărat dacă *x* nu este multiplu de 100 şi zero în caz contrar.

4.

Expresia

getchar()!=EOF

are valoarea adevărat dacă macroul *getchar* a citit un caracter diferit de sfîrşitul de fişier.

5. int x;

scanf("%d",&x)==1

Această expresie are valoarea adevărat dacă *scanf* a citit de la terminalul standard un întreg (scanf returnează numărul cîmpurilor citite de la terminalul standard). În caz contrar, adică dacă la intrare nu se află un întreg, expresia de mai sus are valoarea fals.

3.2.5. Operatorii logici

Aceştia sînt:

! - negaţia logică (operator unar);
&& - şi logic;
|| - sau logic.

Negaţia logică are aceeaşi prioritate ca şi ceilalţi operatori unari ai limbajului C. De altfel, toţi operatorii unari au aceeaşi prioritate care este imediat mai mare decît cea a operatorilor multiplicativi.

Expresia

!*operand*

are valoarea zero (fals) dacă *operand* are o valoare diferită de zero şi

valoarea unu (adevărat) dacă *operand* are valoarea zero.

Operatorul && are prioritatea mai mică decît cei de egalitate. O expresie de forma:

E1&&E2

are valoarea 1, dacă expresiile E1 şi E2 sînt ambele diferite de zero. În rest, expresia de mai sus are valoarea zero.

Operatorul || are prioritatea imediat mai mică decît operatorul &&. Expresia

E1||E2

are valoarea zero, dacă ambele expresii E1 şi E2 la care se aplică operatorul || au valoarea zero. În rest, expresia de mai sus are valoarea unu.

Operatorii logici binari (&& şi ||) se evaluează de la stînga la dreapta. Dacă la evaluarea unei expresii se ajunge într-un punct în care se cunoaşte valoarea de adevăr (fals sau adevărat) a întregii expresii, atunci restul expresiei, aflată în dreapta punctului respectiv, nu se mai evaluează.

Exemple:

1. !a&&b

Dacă *a* are valoarea zero şi *b* este diferit de zero, atunci expresia de mai sus are valoarea unu (adevărat).

Într-adevăr, dacă *a* are valoarea zero, atunci !a are valoarea unu.

Dacă b este diferit de zero, atunci ambii operanzi ai operatorului && sînt diferiţi de zero şi deci întreaga expresie are valoarea unu.

2. !a||b

Această expresie are valoarea zero dacă *a* are valoarea diferită de zero, iar *b* are valoarea zero. În acest caz !a are valoarea zero şi deci ambii operanzi ai operatorului || au valoarea zero. Conform definiţiei operatorului ||, rezultă că întreaga expresie are valoarea zero.

3. !a&&b||!b&&a

Această expresie are valoarea zero dacă *a* şi *b* au ambii valoarea zero sau ambii sînt diferiţi de zero. În caz contrar, expresia are valoarea unu. Acest lucru rezultă din tabela de mai jos:

a	b	!a	!b	!a&&b	!b&&a	!a&&b||!b&&a
0	0	1	1	0	0	0
diferit de zero	diferit de zero	0	0	0	0	0
diferit de zero	0	0	1	0	1	1
0	diferit de zero	1	0	1	0	1

Din cele de mai sus rezultă că expresia

!a&&b||!b&&a

realizează *sau logic exclusiv* între *a* şi *b*.

Pentru a=0 şi *b* diferit de zero, expresia de mai sus se evaluează astfel:

1. Se evaluează !a şi se obţine valoarea unu.
2. Se aplică operatorul *şi logic* la !a şi *b* şi se obţine unu.

Deoarece !a&&b este legat de restul expresiei prin operatorul *sau logic*, rezultă că în acest moment se poate afirma că întreaga expresie este adevărată (are valoarea unu). De aceea, în acest caz nu se mai evaluează restul expresiei, adică termenul !b&&a.

Exerciţii:

3.11 Să se scrie un program care citeşte un număr şi afişează unu dacă numărul respectiv aparţine intervalului [-1000,1000] şi zero în caz contrar.

Dacă notăm cu *a* numărul citit, atunci el trebuie să satisfacă simultan relaţiile:

a >= -1000

şi

a <= 1000

Acest fapt se poate exprima cu ajutorul expresiei:

a >= -1000 && a<=1000.

PROGRAMUL BIII11

```
#include <stdio.h>
main() /* citeste pe a si afiseaza unu daca numarul citit apartine
          intervalului [-1000,1000] si zero in caz contrar */
{
 double a;

 scanf("%lf",&a);
 printf("a=%g\t%d\n", a, a >= -1000 && a <= 1000);
}
```

3.12 Să se scrie un program care citeşte un întreg din intervalul [1600,4900], ce reprezintă un an calendaristic, afişează unu dacă anul este bisect şi zero în caz contrar sau dacă anul nu aparţine intervalului indicat mai sus.

Un an, din calendarul gregorian, este bisect dacă este multiplu de patru şi nu este multiplu de o sută sau dacă este multiplu de patru sute. Această regulă este valabilă pentru anii care nu sînt anteriori anului 1600.

Calendarul gregorian a fost introdus datorită faptului că anul iulian a fost prea lung.

Conform calendarului iulian, un an era bisect, dacă era divizibil cu patru. Din această cauză la 133 de ani se înregistrează o diferenţă de o zi faţă de anul tropic. În felul acesta, în anul 1582 s-a constatat o diferenţă de 10 zile întîrziere şi s-a hotărît corectarea diferenţei respective convenind ca după ziua de joi 4 octombrie 1582 să urmeze ziua de vineri 15 octombrie 1582. De asemenea, s-a stabilit ca durata unui an calendaristic să fie:

365+1/4-1/100+1/400=365,2425 zile

În această expresie termenii au semnificaţia:

365 - Numărul de zile ale anului iulian.

1/4 - Adaosul de un sfert de zi, adică de o zi la patru ani, conform calendarului iulian.

1/100 - Eliminarea unei zile la fiecare o sută de ani. Deci un astfel de an nu mai este bisect, deşi el este multiplu de patru.

1/400 - Adăugarea unei zile la fiecare 400 de ani. În felul acesta, un an multiplu de 400 are o zi în plus, deci este bisect.

Calendarul gregorian, avînd durata anului egală cu 365, 2425, are o eroare de aproximativ 26 de secunde. Aceasta înseamnă că regula indicată mai sus pentru a stabili anii bisecţi (care au 366 de zile) este valabilă pentru

anii următori anului 1582 şi care nu sînt ulteriori anului 4900.

Într-adevăr, eroarea de 26 de secunde dintr-un an gregorian devine o zi abea la aproximativ 3323 de ani, deci după anul 4900 (1582+3323=4905). De aceea, regula de mai sus se poate aplica pentru anii din intervalul [1600,4900].

Dacă notăm cu *an* anul calendaristic, atunci el este bisect dacă de exemplu:

an%4==0 (an este multiplu de 4)

şi

an%100!=0 (an nu este multiplu de o sută).

Deci anul este bisect dacă expresia

(1) an%4==0 && an%100!=0

este adevărată.

De asemenea, anul este bisect şi în cazul în care expresia

(2) an%400==0 (an este multiplu de 400)

este adevărată. Deci anul este bisect dacă este adevărată expresia (1) sau (2), adică dacă este adevărată expresia:

an%4==0&&an%100!=0||an%400==0

PROGRAMUL BIII12

```
#include <stdio.h>
main() /* citeste un intreg care reprezinta un an calendaristic din
          intervalul [1600,4900] si afiseaza 1 daca anul este bisect si
          zero daca nu este sau nu apartine intervalului indicat */
{
 int an;

 scanf("%d", &an);
 printf("an=%d\t%d\n",an,
           an >=1600&&an <= 4900&&(an%4 == 0 &&
           an%100 != 0 || an%400 == 0));
}
```

3.13 Să se scrie un program care citeşte un întreg din intervalul [1600,4900] ce reprezintă un an calendaristic şi afişează numărul de zile din anul respectiv.

PROGRAMUL BIII13

```
#include <stdio.h>
main() /* citeste un intreg care reprezinta un an calendaristic din
          intervalul [1600,4900] si afiseaza numarul de zile din
          anul respectiv */
{
 int an;

 scanf("%d", &an);
 printf("anul %d are %d zile\n", an,
          365 + (an%4 == 0 &&
          an%100 != 0 || an%400 == 0));
}
```

Observaţie:

În acest program se consideră că anul tastat la terminalul standard aparţine intervalului [1600,4900].

3.2.6. Operatorii logici pe biţi

Operatorii logici pe biţi sînt:

~ - complement faţă de unu (operator unar);
<< - deplasare stînga;
>> - deplasare dreapta;
& - şi logic pe biţi;
^ - sau exclusiv logic pe biţi;
| - sau logic pe biţi.

Aceşti operatori se aplică la operanzi de tip întreg. Ei se execută bit cu bit şi operanzii se extind la 16 biţi dacă este necesar.

Complementul faţă de unu are aceeaşi prioritate ca ceilalţi operatori unari. El schimbă fiecare bit 1 al operandului cu zero şi fiecare bit zero al acestuia cu 1.

Exemple:

1. ~1234

1234 se reprezintă în binar astfel:

10011010010

Se extinde la 16 biţi:

0000010011010010

apoi se aplică operatorul de complementare faţă de 1:

1111101100101101

sau în octal:

0175455

2. ~-1234

Operatorii unari au aceeaşi prioritate şi se asociază de la dreapta la stînga, deci întîi se realizează negativarea şi apoi complementarea faţă de 1.

Numărul 1234 se reprezintă în binar:

10011010010

se negativează:

1111101100101110

apoi se complementează faţă de 1:

10011010001

sau în octal:

02321

3. -~1234

În acest caz, întîi se determină complementul faţă de 1, apoi se negativează numărul.

1234 în binar este:

10011010010

se complementează faţă de 1:

1111101100101101

se negativează:

10011010011

sau în octal:

02323

Operatorii de deplasare sînt operatori binari. Ei au aceeaşi prioritate,

care este imediat mai mică decît a operatorilor aditivi şi imediat mai mare decît prioritatea operatorilor relaţionali.

Operatorul << realizează o deplasare la stînga a valorii primului său operand cu un număr de poziţii binare egal cu valoarea celui de-al doilea operand al său. Această operaţie este echivalentă cu înmulţirea cu puteri ale lui 2.

În mod analog, operatorul >> realizează o deplasare la dreapta a valorii primului său operand cu un număr de poziţii binare egal cu valoarea celui de-al doilea operand al său. Această operaţie este echivalentă cu o împărţire cu puteri ale lui 2.

Exemple:

1. int a;

   a<<3

are aceeaşi valoare ca şi

   a*8

adică se înmulţeşte a cu 2 la puterea 3.

2. int a;

   a>>4

are aceeaşi valoare ca şi

   a/16

adică a se împarte la 2 la puterea 4.

3. ~0<<3

   Această expresie se evaluează astfel:

a. Se extinde zero pe 16 biţi:

   0000000000000000

b. Se aplică operatorul de complementare faţă de unu:

   1111111111111111

c. Se fac 3 deplasări spre stînga:

1111111111111000

sau în octal:

177770

4. ~(~0<<3)

Rezultatul din exemplul precedent se complementează faţă de unu:

0000000000000111

Operatorul *şi logic pe biţi* se execută bit cu bit, conform tabelei de mai jos:

1&1=1
1&0=0
0&1=0
0&0=0

Operatorul & are prioritatea imediat mai mică decît operatorii de egalitate. Se utilizează la anulări de biţi.

Exemple:

1. int a;

a&0377

are ca valoare, valoarea octetului mai puţin semnificativ al valorii variabilei *a*.

Într-adevăr, se extinde constanta octală 0377 la 16 biţi:

0000000011111111

apoi se face *şi logic pe biţi* între această constantă şi valoarea variabilei *a*. Se păstrează ultimii 8 biţi ai lui *a*, primii 8 biţi anulîndu-se.

2. int a;

a&0177400

are ca valoare, valoarea lui *a* după ce s-au anulat ultimii 8 biţi ai săi. Aceasta rezultă din faptul că, constanta octală 0177400 are în binar valoarea:

1111111100000000

3. int a;

a&0177776

are ca valoare cel mai mare număr par care nu-l depăşeşte pe *a*.

Diferenţa dintre operatorul *şi logic* (&&) şi operatorul *şi logic pe biţi* (&) constă în aceea că primul se realizează global, faţă de al doilea care se realizează pe biţi.

Aşa de exemplu, dacă x=2 şi y=1, atunci x&&y are valoarea 1, deoarece ambii operanzi sînt diferiţi de zero. În schimb, x&y are valoarea zero. Într-adevăr, în acest caz se realizează un *şi pe biţi* cu valorile:

x=0000000000000010
y=0000000000000001
----
x&y=0000000000000000

Operatorul *sau exclusiv pe biţi* se execută bit cu bit, conform tabelei de mai jos:

1^1=0
1^0=1
0^1=1
0^0=0

Operatorul ^ are prioritatea imediat mai mică decît operatorul &. Se utilizează pentru a anula sau poziţiona diferiţi biţi.

Exemple:

1. int a;

a^a

are valoarea 0. Într-adevăr, cei doi operanzi fiind identici, se execută operatorul ^ pentru biţi identici, ori în acest caz el are ca rezultat pe zero.

2. int a;

a^1

are o valoare care depinde de paritatea lui *a*. Într-adevăr, dacă *a* este par, atunci ultimul său bit este zero.

Expresia

a^1

setează la valoarea unu ultimul bit a lui *a*. În felul acesta, rezultatul este egal cu cel mai mic număr impar care-l depăşeşte pe *a*.

Dacă *a* este impar, atunci ultimul bit are valoarea unu.

Expresia

a^1

anulează ultimul bit a lui *a*. Deci, rezultatul este egal cu cel mai mare număr par care nu-l depăşeşte pe *a*.

Operatorul *sau logic pe biţi* se execută bit cu bit, conform tabelei de mai jos:

1|1=1
1|0=1
0|1=1
0|0=0

Acest operator are prioritatea imediat mai mică decît operatorul *sau exclusiv pe biţi* (^) şi imediat mai mare decît operatorul *şi logic* (&&).

Operatorul | se foloseşte la setări de biţi.

Exemple:

1. int a;

a|1

are o valoare a cărui ultim bit este unu, indiferent de valoarea variabilei *a*. Ceilalţi biţi coincid cu cei ai lui *a*. De aceea, valoarea acestei expresii este cel mai mic număr impar care nu este mai mic decît *a*.

2. int a;

a|0100000

are o valoare cu bitul cel mai semnificativ setat, indiferent de valoarea variabilei *a*. Ceilalţi biţi coincid cu cei ai lui *a*.

3.2.7. Operatorii de atribuire

Operatorul de atribuire, în forma cea mai simplă, se notează prin caracterul "=". El se utilizează în expresii de forma:

v=(*expresie*)

unde:

v - Este o variabilă simplă, referenţiază un element de tablou (variabilă cu indici) sau de structură.

Operatorul de atribuire are o prioritate mai mică decît toţi operatorii pe care i-am întîlnit pînă în prezent. De aceea, parantezele din construcţia indicată mai sus, de obicei nu sînt necesare.

Operatorul de atribuire are ca efect atribuirea valorii expresiei aflată în dreapta semnului de atribuire variabilei *v*. Ulterior o să vedem că în stînga semnului de atribuire se poate afla chiar o expresie care defineşte o adresă.

La o atribuire se face şi o conversie dacă este necesar. Astfel, valoarea expresiei din dreapta semnului de atribuire se va converti spre tipul variabilei din stînga lui înainte de a se face atribuirea, dacă cele două tipuri sînt diferite.

Expresia

v=expresie

este o *expresie de atribuire*. Ea are ca *valoare* valoarea care se atribuie, iar ca *tip*, tipul variabilei *v*.

Rezultă că o construcţie de forma:

v1=(v=expresie)

este corectă şi reprezintă tot o expresie de atribuire: lui *v1* i se atribuie valoarea atribuită în prealabil lui *v*, făcîndu-se şi conversie dacă este necesar.

Deoarece operatorii de atribuire se asociază de la *dreapta spre stînga*, expresia de mai sus se poate scrie fără paranteze:

v1=v=expresie

În general, o expresie de atribuire are forma:

vn=...=v1=v=expresie

Valoarea expresiei aflate în partea dreaptă se atribuie întîi lui v, apoi lui v1 şi aşa mai departe şi în final se atribuie lui *vn*. Atribuirile sînt precedate de conversii în cazul în care valoarea care se atribuie este de un tip diferit decît tipul variabilei la care se face atribuirea.

Pentru operatorii de atribuire, în afara semnului = se mai poate folosi şi succesiunea de caractere:

op=

unde *op* este un operator binar aritmetic sau logic pe biţi. Deci *op* poate fi unul din operatorii:

/ % * - + << >> & ^ |

Această construcţie se foloseşte pentru a face prescurtări.

Expresia

v op= expresie

este echivalentă cu expresia de atribuire:

v=v op(expresie)

Expresiile de atribuire pot fi folosite peste tot în program unde este legal să apară o expresie. Acest fapt permite adesea să se facă compactări în programul sursă.

Exemple:

1.

```
int a;
a=10
```

variabilei *a* i se atribuie valoarea 10.

2.

```
int i;
i=i+3
```

valoarea variabilei *i* se măreşte cu 3.

3.

```
int i;
i += 3
```

are acelaşi efect ca şi expresia de la exemplul precedent.

4. Expresia de atribuire:

```
a[i*3+10][j*2-3]=a[i*3+10][j*2-3]*x
```

se scrie prescurtat astfel:

```
a[i*3+10][j*2-3]*=x
```

Exerciţii:

3.14 Să se scrie un program care citeşte valoarea lui *x*, calculează valorile expresiilor:

4x*x+3x

4x*x+3x+1

şi

(4x*x+3x-1)/(4x*x+3x+1)

şi afişează valorile respective.

PROGRAMUL BIII14

```
#include <stdio.h>
main() /* citeste pe x si afiseaza valorile expresiilor:
            4x*x + 3x
            4x*x + 3x +1
            (4x*x + 3x - 1 )/(4x*x + 3x +1) */
{
 double x,y;

 scanf("%lf", &x);
 printf("x=%g\ty=4x*x + 3x=%g\n",x,y=4*x*x + 3*x );
 printf("4x*x+3x+1 = %g\t (4x*x+3x-1)/(4x*x+3x+1)\
         = %g\n",y+1, (y-1)/(y+1));
}
```

Observaţie:

Parametrul:

y=4*x*x+3*x

din primul apel al funcţiei *printf* este o expresie de atribuire. Prin intermediul ei se atribuie lui y valoarea expresiei

4*x*x+3*x

Valoarea expresiei de atribuire coincide cu valoarea atribuită lui *y*. *Tipul expresiei* de atribuire coincide cu tipul lui *y*, deci tipul ei este *double*.

3.15 Să se scrie un program care citeşte valoarea lui *x*, calculează şi afişează valorile expresiilor:

x**10, x**20, x**30 şi x**40.

PROGRAMUL BIII15

```
#include <stdio.h>
#include <math.h>

main() /* citeste pe x si afiseaza valorile expresiilor:
                x**10
                x**20
                x**30
                x**40 */
{
 double x,y,z;

 scanf("%lf", &x);
 printf("x=%g\tx**10 =%g\n", x, y = pow(x,10.0));
 printf("\tx**20 =%g\n", z= y*y);
 printf("\tx**30 =%g\n", y*z);
 printf("\tx**40 =%g\n", z*z);
}
```

3.16 Să se scrie un program care citeşte valoarea lui *x*, afişează valorile lui *x*, [x], {x}, calculează şi afişează valorile expresiilor:

7[x]*[x]-3[x]+10

şi

7{x}*{x}-3{x}+10

unde:

[x] - Notează partea întreagă a lui *x* ([x]<=x).
{*x*} - Notează partea fracţionară a lui *x* ({x}=x-[x]).

PROGRAMUL BIII16

```
#include <stdio.h>
main() /* - citeste pe x;
          - afiseaza pe x, [x], {x};
          - calculeaza si afiseaza valorile expresiilor:
                7[x]*[x] - 3[x] + 10
            si
                7{x}*{x} - 3{x} + 10 */
{
 int i;
 double x,y;
```

```
    scanf("%lf", &x);
    printf("x=%g\t",x);
    printf("[x]=%d\t",i = x);
    printf("{x}=%f\n",y = x -i);
    printf("7[x][x] - 3[x] + 10=%d\n",
           7*i*i - 3*i +10);
    printf("7{x}{x} - 3{x} + 10=%g\n",
           7*y*y - 3*y +10);
}
```

3.2.8. Operatorii de incrementare şi decrementare

Aceşti operatori sînt unari şi au aceeaşi prioritate ca şi ceilalţi operatori unari ai limbajului C.

Operatorul de *incrementare* se notează cu ++, iar cel de *decrementare* cu --.

Operatorul de incrementare măreşte valoarea operandului său cu 1, iar cel de decrementare micşorează valoarea operandului său cu 1.

Aceşti operatori pot fi folosiţi *prefixaţi*:

++*operand*
--*operand*

sau *postfixaţi*:

operand++
operand--

În cazul în care un operator de incrementare sau decrementare este prefixat, se foloseşte valoarea operandului la care s-a aplicat operatorul respectiv.

În cazul în care un operator de incrementare sau decrementare este postfixat, se foloseşte valoarea operandului dinaintea aplicării operatorului respectiv.

Exemplu:

Presupunem că *x* are valoarea 3. Dacă considerăm expresia de atribuire

y=++x

atunci lui *y* i se atribuie valoarea 4 (la atribuire se foloseşte valoarea incrementată).

Dacă utilizăm expresia de atribuire

```
y=x++
```

pentru x=3, atunci lui *y* i se atribuie valoarea 3 (la atribuire se foloseşte valoarea dinaintea incrementării).

În ambele cazuri valoarea lui *x* s-a mărit cu 1.

3.2.9. Operatorul de forţare a tipului sau de conversie explicită (expresie cast)

Adesea dorim să specificăm conversia valorii unui operand spre un *tip dat.* Acest lucru este posibil folosind o construcţie de forma:

(tip) operand

Printr-o astfel de construcţie valoarea operandului se converteşte spre tipul indicat în paranteze.

În construcţia de mai sus (tip) se consideră că este un operator unar. Îl vom numi *operator de forţare a tipului* sau de *conversie explicită.*

Construcţia de mai sus o vom numi *expresie cast.*

Exemplu:

Presupunem că funcţia *f* are un parametru de tip *double*. Fie declaraţia:

```
int n;
```

Atunci, pentru a apela *f* cu parametrul *n*, este necesar ca, în prealabil, *n* să se convertească spre *double*. Acest lucru se poate realiza printr-o atribuire:

```
double x;
...
f(x=n);
```

Un alt mod mai simplu de conversie a parametrului întreg spre tipul *double* este utilizarea unei *expresii cast*:

```
f((double)n);
```

Operatorul *(tip)* fiind unar, are aceeaşi prioritate ca şi ceilalţi operatori unari ai limbajului C.

Exerciţii:

3.17 Să se scrie un program care citeşte un întreg şi afişează rădăcina pătrată din numărul respectiv.

PROGRAMUL BIII17

```
#include <stdio.h>
#include <math.h>

main() /* - citeste pe n;
          - calculeaza si afiseaza radacina patrata din n */
{
 long n;

 scanf("%ld", &n);
 printf("n=%ld\tsqrt(n)=%g\n",n,sqrt((double)n));
}
```

Observaţie:

În acest program s-a utilizat funcţia *sqrt* pentru extragerea rădăcinii pătrate. Ea are prototipul:

double sqrt(double);

Acest prototip este definit în fişierul *math.h*.

2.18 Să se scrie un program care citeşte pe *n* de tip întreg şi afişează valoarea expresiei n/(n+1) cu 15 zecimale.

PROGRAMUL BIII18

```
#include <stdio.h>
main() /* - citeste pe n;
          - calculeaza si afiseaza pe n/(n+1) cu 15 zecimale */
{
 long n;

 scanf("%ld", &n);
 printf("n=%ld\tn/(n+1)=%.15g\n", n,
        (double)n/(n+1));
}
```

Observaţie:

Expresia n/(n+1) realizează împărţirea întreagă a lui *n* la *n+1*. Pentru a obţine cîtul împărţirii cu 15 zecimale este necesar să se efectueze împărţirea neîntreagă. În acest scop s-a convertit operandul *n* spre tipul *double*. În felul

acesta, conform regulei conversiilor implicite, se converteşte spre *double* şi cel de al doilea operand şi apoi se face împărţirea celor doi operanzi flotanţi.

3.2.10. Operatorul dimensiune

Dimensiunea în octeţi a unei date sau al unui tip se poate determina folosind operatorul *sizeof*. El poate fi folosit sub forma:

sizeof *data*

sau

sizeof(*tip*)

unde:

data - Poate fi:

- un nume de variabilă simplă;
- un nume de tablou;
- un nume de structură;
- referirea la un element de tablou (variabilă cu indici);
- referirea la elementul unei structuri.

tip - Poate fi:

- un cuvînt sau cuvintele cheie ale unui tip predefinit;
- o construcţie care defineşte un tip.

Expresia

sizeof data

poate fi scrisă şi sub forma sizeof(data). Ea are ca valoare dimensiunea în octeţi a operandului "data". Astfel, dacă data este:

numele unei variabile simple - Atunci rezultatul va fi numărul de octeţi alocaţi variabilei respective.

numele unui tablou - Atunci rezultatul va fi numărul de octeţi al zonei de memorie alocate tabloului respectiv.

numele unei structuri - Atunci rezultatul va fi numărul de octeţi al zonei de memorie alocate structurii respective.

etc.

Expresia

sizeof(tip)

are ca valoare numărul de octeţi alocaţi pentru reprezentarea unei date de tip *tip*.

Operatorul dimensiune (*sizeof*) este unar şi are aceeaşi prioritate ca restul operatorilor unari ai limbajului C.

Exemple:

1.

```
int x;
sizeof x
```

are valoarea 2, deoarece pentru variabila *x* se alocă 2 octeţi.

2.

```
long double y[7];
sizeof y[3]
```

are valoarea 10, deoarece pentru o dată de tip *long double* se alocă 10 octeţi.

```
sizeof y
```

are valoarea 70, deoarece tabloul are 7 elemente în total.

3.

```
sizeof(char)
```

are valoarea 1, deoarece pentru o dată de tip *char* se alocă un octet.

3.2.11. Operatorul adresă

Operatorul adresă este unar şi se notează prin caracterul **&**. El se aplică pentru a determina adresa de început a zonei de memorie alocată unei date. În forma cea mai simplă, acest operator se utilizează în construcţii de forma:

&nume

unde:

nume - Este numele unei variabile simple sau al unei structuri.

Menţionăm că în cazul în care *nume* este numele unui tablou, atunci acesta are ca valoare chiar adresa de început a zonei de memorie alocată

tabloului respectiv. Deci, în acest caz nu se mai utilizează operatorul adresă &.

Acest operator a fost utilizat deja frecvent la apelul funcţiei *scanf*.

Mai tîrziu o să vedem şi alte utilizări ale acestui operator.

Fiind un operator unar, el are aceeaşi prioritate ca şi ceilalţi operatori unari ai limbajului C.

Operatorul unar & are în limbajul C++ şi o altă utilizare aşa cum se va vedea mai tîrziu.

3.2.12. Operatorii paranteză

Parantezele rotunde se utilizează fie pentru a include o expresie, fie la apelul funcţiilor.

Amintim că o expresie inclusă în paranteze rotunde formează un operand. În felul acesta se poate impune o altă ordine în efectuarea operaţiilor, decît cea care rezultă din prioritatea şi asociativitatea operatorilor.

Operanzii obţinuţi prin includerea unei expresii între paranteze impun anumite limite asupra operatorilor. De exemplu, la un astfel de operand nu se pot aplica operanzii de incrementare şi decrementare sau operatorul adresă. Astfel construcţiile:

(i+10)++ --(x+y) &(x+y)

sînt eronate.

La apelul unei funcţii, lista parametrilor efectivi se include între paranteze rotunde. În acest caz se obişnuieşte să se spună că parantezele rotunde sînt *operatori de apel de funcţie*.

Parantezele pătrate includ expresii care reprezintă *indici*. Ele se numesc *operatori de indexare*.

Parantezele sînt operatori de prioritate maximă. Operatorii unari au prioritatea imediat mai mică decît parantezele.

3.2.13. Operatorii condiţionali

Operatorii condiţionali permit construirea de expresii a căror valoare să depindă de valoarea unei condiţii.

Prin *condiţie* înţelegem o expresie care poate avea două valori: *adevărat* sau *fals*. În limbajul C, o condiţie se reprezintă printr-o expresie oarecare.

Ea are valoarea *adevărat* dacă este diferită de zero şi *fals* în caz contrar.

Numim *expresie condiţională*, o expresie a cărei valoare şi tip este dependentă de valoarea unei condiţii.

Un exemplu simplu de expresie condiţională este aceea care are ca valoare maximul dintre două numere. Astfel, dacă *a* şi *b* sînt două numere, atunci valoarea expresiei care calculează maximul dintre *a* şi *b* depinde de valoarea condiţiei a>b.

Într-adevăr, dacă a > b are valoarea adevărat, atunci expresia respectivă are valoarea *a*. În caz contrar (adică a>b are valoarea fals) expresia de calcul a maximului va avea valoarea *b*.

O expresie condiţională are formatul:

E1?E2:E3

unde:

E1,E2 şi E3 - Sînt expresii.

Această expresie se evaluează astfel:

- Se determină valoarea expresiei *E1*.
- Dacă *E1* are o valoare diferită de zero (are valoarea adevărat), atunci valoarea şi tipul expresiei condiţionale coincide cu valoarea şi tipul expresiei *E2*. Altfel (adică *E1* are valoarea zero) valoarea şi tipul expresiei condiţionale coincide cu valoarea şi tipul expresiei *E3*.

De aici rezultă că valoarea şi tipul expresiei condiţionale depinde de valoarea expresiei *E1*.

Folosind formatul de mai sus, calculul maximului dintre *a* şi *b* se exprimă cu ajutorul expresiei condiţionale:

(a > b) ? a : b

unde:

(a > b) - Este expresia E1 (condiţia).
a - Este expresia E2.
b - Este expresia E3.

Operatorii condiţionali sînt:

? şi :

Ei se folosesc totdeauna împreună şi în ordinea indicată în formatul expresiei condiţionale.

Ei au prioritatea imediat mai mică decît prioritatea operatorului *sau logic* (||) şi imediat mai mare decît prioritatea operatorilor de atribuire. Ţinînd seama de prioritatea operatorilor, rezultă că se pot omite parantezele din expresia pentru calculul maximului dintre *a* şi *b*:

a > b ? a : b

Expresia condiţională este un caz particular de expresie şi deci ea poate fi utilizată oriunde este legal să apară, în program, o expresie.

De exemplu:

v = (*E1* ? *E2* : *E3*)

este o expresie de atribuire: în partea dreaptă a semnului de atribuire se află o expresie condiţională. Ea poate fi scrisă mai simplu fără paranteze, deoarece operatorul de atribuire este mai puţin prioritar decît operatorii condiţionali.

Operatorii condiţionali se asociază de la dreapta la stînga la fel ca şi operatorii unari şi de atribuire.

Exerciţii:

3.19 Să se scrie un program care citeşte două numere şi afişează maximul dintre ele.

PROGRAMUL BIII19

```
#include <stdio.h>
main() /* citeste doua numere si afiseaza maximul dintre ele */
{
 double a,b;

 scanf("%lf %lf", &a,&b);
 printf("a=%g\tb=%g\tmax(a,b)=%g\n", a,b,
        a > b ? a : b );
}
```

3.20 Să se scrie un program care citeşte un număr şi afişează valoarea lui absolută.

PROGRAMUL BIII20

```
#include <stdio.h>
main() /* citeste un numar si afiseaza valoarea lui absoluta */
{
```

```
 double a;

 scanf("%lf", &a);
 printf("a=%g\tabs(a)=%g\n",a, a < 0 ? -a : a );
}
```

3.21 Să se scrie un program care citeşte doi întregi şi afişează maximul dintre valorile lor absolute.

PROGRAMUL BIII21

```
#include <stdio.h>
main() /* citeste doua numere intregi si afiseaza maximul dintre
            valorile lor absolute */
{
 int a,b,c,d;

 scanf("%d %d", &a,&b);
 printf("a=%d\tb=%d\tabs(a)=%d\tabs(b)=%d\n",a,b,
        c =a<0 ? -a: a, d=b < 0 ? -b : b);
 printf("max(abs(a),abs(b))=%d\n",c > d ? c : d );
}
```

3.22 Să se scrie un program care citeşte valoarea variabilei *x* şi afişează valoarea funcţiei *f(x)* definită ca mai jos:

$$f(x)=\begin{cases} 3x*x+7x-10 & \text{pentru } x<0; \\ 2 & \text{pentru } x=0; \\ 4x*x-8 & \text{pentru } x>0. \end{cases}$$

PROGRAMUL BIII22

```
#include <stdio.h>
main() /* citeste pe x si afiseaza valoarea functiei f(x) definita astfel:
                  3x*x + 7x - 10    pentru x < 0;
                  2                 pentru x = 0;
                  4x*x - 8          pentru x > 0 */
{
 double x;

 scanf("%lf", &x);
 printf("x=%g\tf(x)=%g\n",x,
        x<0 ? 3*x*x+7*x-10 : x>0 ? 4*x*x-8:2);
}
```

3.2.14. Operatorul virgulă

Operatorul virgulă leagă două expresii în una singură conform formatului de mai jos:

exp1,exp2

Operatorul virgulă are cea mai mică prioritate dintre toţi operatorii limbajului C. Prioritatea lui este imediat mai mică decît a operatorilor de atribuire.

Construcţia de mai sus este o expresie. Valoarea şi tipul acestei expresii coincide cu valoarea şi tipul ultimei expresii, deci în cazul de faţă cu a lui exp2.

O construcţie de forma:

(exp1,exp2),exp3

este corectă şi ea este o nouă expresie. Într-adevăr, *exp1, exp2* fiind expresii, *(exp1,exp2)* este un operand, deci tot o expresie, aşa că după ea poate fi scrisă o virgulă urmată de o altă expresie. Deoarece operatorul virgulă are prioritatea cea mai mică, rezultă că parantezele pot fi omise, deci expresia de mai sus se poate scrie astfel:

exp1,exp2,exp3

În general, o construcţie de forma:

exp1,exp2,...,expn

unde:

exp1,exp2,...expn - Sînt expresii.

este o expresia a cărei valoare şi tip coincide cu valoarea şi tipul lui *expn*.

Într-o astfel de construcţie, expresiile se evaluează pe rînd, de la stînga la dreapta.

Operatorul virgulă se utilizează în situaţii în care într-un anumit punct al unui program în care este legal să folosim o expresie, este necesar să se realizeze un calcul complex exprimat prin mai multe expresii.

Exerciţii:

3.23 Să se scrie un program care citeşte doi intregi şi afişează maximul dintre valorile lor absolute.

Acest exerciţiu a fost rezolvat şi în paragraful precedent (exerciţiul 3.21.)

PROGRAMUL BIII23

```
#include <stdio.h>
main() /* citeste doi intregi si afiseaza maximul dintre valorile
          lor absolute */
{
 int a,b,c,d;

 scanf("%d%d", &a,&b);
 printf("a=%d\tb=%d\tmax(abs(a),abs(b))=%d\n",
        a,b,((c=a<0?-a:a),
        (d=b<0 ? -b : b), (c > d)) ? c : d );
}
```

Observaţie:

Expresia:

(c=a<0?-a),(d=b<0?-b:b),(c>d)

se compune din trei expresii care se evaluează de la stînga la dreapta.

Prima atribuie lui *c* valoarea absolută a lui *a*, a doua atribuie lui *d* valoarea absolută a lui *b*, iar a treia testează relaţia c>d. Valoarea întregii expresii coincide cu 1 dacă c>d şi cu zero în caz contrar.

3.2.15. Alţi operatori ai limbajului C

În limbajul C se utilizează şi alţi operatori şi anume:

- operatorul * unar;

şi

- operatorii . şi ->.

Operatorul * unar se utilizează pentru a face acces la conţinutul unei zone de memorie definită prin adresa ei de început. Se obişnuieşte să se spună că operatorul adresă (& unar) este *operator de referenţiere*, iar operatorul * unar este *operator de dereferenţiere*. Acest operator va fi studiat ulterior împreună cu datele de tip adresă (*pointer*). El are aceeaşi prioritate ca şi ceilalţi operatori unari ai limbajului C.

Operatorii . şi -> (acest simbol este compus din semnul "minus" urmat de semnul "mai mare") se utilizează pentru a se face acces la componentele unei structuri. Ei sînt de prioritate maximă, avînd aceeaşi prioritate cu parantezele. Ei vor fi studiaţi în capitolul cu privire la structuri.

3.2.16. Tabela cu priorităţile operatorilor limbajului C

În tabela de mai jos se indică operatorii limbajului C în ordinea descrescătoare a priorităţilor lor.

Operatorii din aceeaşi linie au aceeaşi prioritate. Ei se asociază de la stînga spre dreapta, exceptînd operatorii unari, condiţionali şi de atribuire, care se asociază de la dreapta spre stînga.

()	[]	.	->							
+(unar)	-(unar)	&(unar)	*(unar)	++	--	(tip)	sizeof	!	~	
*(binar)	/	%								
+(binar)	-(binar)									
<<	>>									
<	<=	>=	>							
==	!=									
&(binar)										
^										
|										
&&										
||										
?	:									
=	<<=	>>=	+=	-=	*=	/=	%=	&=	^=	|=
,										

Ulterior această tabelă va fi completată şi cu alţi operatori specifici limbajului C++.

4. INSTRUCŢIUNI

Am văzut că o funcţie are structura:

antet
corp

Corpul unei funcţii conţine, între acolade, o succesiune de declaraţii urmate de o succesiune de instrucţiuni:

{
declaraţii
instrucţiuni
}

Corpul poate conţine numai instrucţiuni sau numai declaraţii, sau se poate reduce la cele două acolade (acoladele vor fi totdeauna prezente).

Declaraţiile permit utilizatorului să definească date de diferite tipuri (predefinite sau definite de utilizator).De asemenea, datele pot fi iniţializate cu ajutorul declaraţiilor.

Prelucrarea datelor se realizează cu ajutorul *instrucţiunilor*.

Ordinea în care se execută instrucţiunile unui program defineşte aşa numita *structură de control a programului*.

Cea mai simplă structură de control este structura *secvenţială*. O astfel de structură de control se compune dintr-o succesiune de instrucţiuni care se execută una după alta, în ordinea în care sînt scrise în program.

De obicei, la descrierea unui proces de calcul este nevoie să se utilizeze şi alte tipuri de structuri. Astfel, C. Bohm şi G. Jacopini au arătat în lucrarea [1], că pentru exprimarea proceselor de calcul sînt suficiente trei structuri de control şi anume:

- structura secvenţială;
- structura alternativă;

şi

- structura repetitivă (ciclică) condiţionată anterior.

Acest rezultat s-a aflat la baza ideii care a condus în anii '70 la conceptul de *programare structurată*. Acest concept a fost dezvoltat de E.W. Dijkstra în lucrările sale, iar ulterior şi de alţi specialişti, ca de exemplu, N. Wirth şi C.A.R. Hoare. El reprezintă un *stil* în programare care se impune şi în prezent. Un efect imediat al programării structurate este ridicarea produc-

tivității în programare și creșterea fiabilității programelor.

Prin *program structurat* înțelegem un program care are o structură de control realizată numai cu ajutorul celor trei structuri amintite mai sus.

Ulterior s-au admis încă două structuri și anume:

- structura selectivă;

și

- structura repetitivă (ciclică) condiționată posterior.

Introducerea acestor structuri permite o flexibilitate mai mare în programare. În acest fel, programarea structurată reprezintă un stil în programare care contribuie la realizarea de programe care au o structură clară și care pot fi ușor depanate și întreținute.

Limbajul C a fost prevăzut cu instrucțiuni menite să permită realizarea simplă a structurilor proprii programării structurate. Structura secvențială se realizează cu ajutorul instrucțiunii *compuse*, structura alternativă cu ajutorul instrucțiunii *if*, structura repetitivă condiționată anterior, prin intermediul instrucțiunilor *while* și *for*, structura selectivă se realizează cu ajutorul instrucțiunii *switch*, iar structura repetitivă condiționată posterior cu ajutorul instrucțiunii *do-while*.

Limbajul C are și alte instrucțiuni care reprezintă elemente de bază în construirea structurilor amintite mai sus. Astfel de instrucțiuni sînt: instrucțiunea *expresie* și instrucțiunea *vidă*.

Alte instrucțiuni prezente în limbaj asigură o flexibilitate mare în programare. Acestea sînt instrucțiunile:

return, break, continue și *goto*.

Mai jos se descriu aceste instrucțiuni și se dau exemple simple de utilizare.

4.1. Instrucțiunea vidă

Instrucțiunea vidă se reduce la caracterul punct și virgulă (;). Ea nu are nici un efect.

Instrucțiunea vidă se utilizează în construcții în care se cere prezența unei instrucțiuni, dar nu trebuie să se execute nimic în punctul respectiv. Astfel de situații apar frecvent în cadrul structurii alternative și repetitive, așa cum se va vedea în continuare.

4.2. Instrucţiunea expresie

Instrucţiunea expresie se obţine scriind punct şi virgulă după o expresie. Deci, instrucţiunea expresie are formatul:

expresie;

În cazul în care expresia din compunerea unei instrucţiuni expresie este o expresie de atribuire, se spune că instrucţiunea respectivă este o *instrucţiune de atribuire*.

Un alt caz utilizat frecvent este acela cînd expresia este un operand ce reprezintă apelul unei funcţii. În acest caz, instrucţiunea expresie este o *instrucţiune de apel* a funcţiei respective.

Exemple:

1.

```
int x;
...
x=10;
```

Este o instrucţiune de atribuire. Variabilei *x* i se atribuie valoarea 10.

2.

```
double y;
...
y=y+4; sau y+=4;
```

Este o instrucţiune de atribuire, care măreşte cu 4 valoarea lui *y*.

3.

```
putch(c-'a'+'A');
```

Este o instrucţiune de apel a funcţiei *putch*.

4.

```
double a;
...
a++;
```

Este o instrucţiune expresie, care măreşte valoarea lui *a* cu unu.

5.

double a;

...

++a;

Are acelaşi efect ca şi instrucţiunea expresie din exemplul precedent.

Observaţie:

Nu orice expresie urmată de punct şi virgulă formează o instrucţiune expresie efectivă. De exemplu, construcţia:

a;

deşi este o instrucţiune expresie, ea nu are nici un efect.

Exerciţii:

4.1 Să se scrie un program care citeşte trei întregi şi afişează maximul dintre ei.

PROGRAMUL BIV1

```
#include <stdio.h>
main() /* citeste trei intregi si afiseaza maximul dintre ei */
{
 int a,b,c,d;

 scanf("%d %d %d", &a,&b,&c);
 d = a > b ? a : b;      // d=max(a,b)
 printf("a=%d\tb=%d\tc=%d\tmax(a,b,c)=%d\n",a,b,
         c,d>c ? d: c);
}
```

4.2 Să se scrie un program care citeşte valorile variabilelor *a*, *b*, *c*, *d*, *x* de tip *double* şi afişează valoarea expresiei:

(a*x*x+b*x+c)/(a*x*x*x+b*x+d)

dacă numitorul este diferit de zero şi zero în caz contrar.

PROGRAMUL BIV2

```
#include <stdio.h>
main() /* citeste pe a,b,c,d si x, calculeaza si afiseaza
          valoarea expresiei:
              (a*x*x+b*x+c)/(a*x*x*x+b*x+d)
```

daca numitorul este diferit de zero si zero in caz contrar */

```
{
 double a,b,c,d,x;
 double y,z,u;

 printf("a=");
 scanf("%lf",&a);
 printf("b=");
 scanf("%lf", &b);
 printf("c=");
 scanf("%lf",&c);
 printf("d=");
 scanf("%lf",&d);
 printf("x=");
 scanf("%lf",&x);
 y = a*x*x;
 z=b*x;
 u=y*x+z+d;
 printf("(a*x*x+b*x+c)/(a*x*x*x+b*x+d)=%g\n",
          u ? (y+z+c)/u : u);
}
```

4.3. Instrucţiunea compusă

Instrucţiunea compusă este o succesiune de instrucţiuni incluse între acolade, succesiune care poate fi precedată şi de declaraţii:

{
declaraţii
instrucţiuni
}

Dacă declaraţiile sînt prezente, atunci ele definesc variabile care sînt definite atît timp cît controlul programului se află la o instrucţiune din compunerea instrucţiunii compuse.

Exemplu:

Presupunem că într-un anumit punct al programului este necesar să se permute valorile variabilelor întregi a şi b. Aceasta se poate realiza astfel:

```
{
 int t;

 t=a;
```

```
   a=b;
   b=t;
}
```

Variabila *t* este definită de îndată ce controlul programului ajunge la prima instrucţiune din instrucţiunea compusă (t=a;). După execuţia ultimei instrucţiuni a instrucţiunii compuse, variabila *t* nu mai este definită.

Instrucţiunea compusă se utilizează unde este necesară prezenţa unei instrucţiuni dar procesul de calcul din punctul respectiv este mai complex şi se exprimă prin mai multe instrucţiuni. În acest caz instrucţiunile respective se includ între acolade pentru a forma o instrucţiune compusă.

Acest procedeu de a forma o instrucţiune compusă din mai multe instrucţiuni, se utilizează frecvent în construirea structurilor alternative şi ciclice.

4.4. Instrucţiunea if

Instrucţiunea *if* are următoarele formate:

format1

if(*expresie*)

instrucţiune

format2

if(*expresie*)

instrucţiune1

else

instrucţiune2

La întîlnirea instrucţiunii *if* întîi se evaluează expresia din paranteze. Apoi, în cazul formatului 1, dacă expresia are valoarea diferită de zero (adică are valoarea adevărat), atunci se execută *instrucţiune*; altfel se trece în secvenţă la instrucţiunea următoare instrucţiunii *if*. În cazul formatului 2, dacă expresia are o valoare diferită de zero, atunci se execută instrucţiune 1 şi apoi se trece în secvenţă (adică la instrucţiunea aflată după instrucţiune 2); altfel se execută instrucţiune 2.

În mod normal, în ambele formate după execuţia instrucţiunii *if* se ajunge la instrucţiunea următoare ei. Cu toate acestea, este posibilă şi o altă situaţie cînd instrucţiunile din compunerea lui *if*, definesc ele însele un alt mod de continuare a execuţiei programului.

Deoarece o instrucţiune compusă este considerată ca fiind un caz particular de instrucţiune, rezultă că instrucţiunile din compunerea lui *if* pot fi instrucţiuni compuse. De asemenea, instrucţiunile respective pot fi chiar instrucţiuni *if*. În acest caz se spune că instrucţiunile *if* sînt *imbricate*.

Exerciţii:

4.3 Se dă funcţia:

y=3*x*x+2*x-10 pentru x>0

şi

y=5*x+10 pentru x<=0

Să se scrie un program care citeşte valoarea lui *x*, calculează şi afişează valoarea lui *y*. Acest proces de calcul implică două alternative, în funcţie de valoarea lui *x*:

dacă x>0, atunci y=3*x*x+2*x-10
altfel (adică x<=0), atunci y=5*x+10.

Aceasta se transcrie imediat, în limbajul C, folosind formatul 2 al instrucţiunii *if*:

```
if(x>0)              /* x>0 este expresie din formatul 2 */
 y=3*x*x+2*x-10;  /* instrucţiune 1 */
else
 y=5*x+10;           /* instrucţiune 2 */
```

Acelaşi proces de calcul se poate realiza cu ajutorul expresiei de atribuire:

y=x>0?3*x*x+2*x-10:5*x+10;

care utilizează în partea dreaptă o expresie condiţională. Aceasta este o scriere mai compactă, dar nu atît de evidentă, ca şi instrucţiunea *if* de mai sus.

PROGRAMUL BIV3

```
#include <stdio.h>
main() /* citeste pe x, calculeaza si afiseaza valoarea lui
          y definita astfel:
                   y = 3x*x +2*x - 10   daca x > 0
                   y = 5x + 10          daca x <= 0 */
{
 float x,y;
```

```
 scanf("%f",&x);
 if(x > 0)
    y= 3*x*x + 2*x -10;
 else
    y= 5*x + 10;
 printf("x= %f\ty= %f\n", x,y);
}
```

4.4 Să se scrie un program care citeşte valoarea lui x, calculează şi afişează valoarea lui y definită ca mai jos:

y=4*x**3+5*x*x-2*x+1 pentru x<0;

y=100 pentru x=0;

y=2*x*x+8*x-1 pentru x>0.

Acest proces de calcul implică următoarele alternative:

dacă x < 0 atunci y=4*x**3+5*x*x-2*x+1

altfel

dacă x=0 atunci y=100

altfel

y=2*x*x+8*x-1

El se transcrie imediat printr-o instrucţiune if *imbricată*:

```
if(x<0)              /* x<0 este expresie din formatul 2 */
 y=4*x**3+5*x*x-2*x+1; // instrucţiune 1
else
// instrucţiune 2 este instrucţiune if
 if(x==0)
    y=100;
 else
    y=2*x*x+8*x-1;
```

Acelaşi proces de calcul se poate descrie prin instrucţiunea expresie de mai jos:

y=x<0?4*x**3+5*x*x-2*x+1:x==0?100:2*x*x+8*x-1;

PROGRAMUL BIV4

```
#include <stdio.h>
main() /* citeste pe x, calculeaza si afiseaza pe y definit astfel:
```

```
                    y = 4*x*x*x + 5*x*x - 2*x +1  daca x<0;
                    y = 100                        daca x=0;
                    y = 2*x*x + 8*x -1            altfel */
{
 float x,y,a;

 scanf("%f", &x);
 a = x*x;
 if( x < 0 )
    y = 4*x*a + 5*a - 2*x +1;
 else
    if( x == 0)
       y = 100;
    else
       y = 2*a + 8*x -1;
 printf("x=%f\ty=%f\n", x,y);
}
```

4.5 Să se scrie un program care citeşte valorile variabilelor neîntregi *a* şi *b*, calculează rădăcina ecuaţiei:

ax+b=0

şi afişează rezultatul.

Procesul de calcul al rădăcinii ecuaţiei de mai sus trebuie să verifice existenţa ei:

dacă *a* este diferit de zero

atunci *x=-b/a*

altfel

dacă *b=0*

atunci ecuaţia este *nedeterminată*;

altfel ecuaţia *nu are soluţie*.

PROGRAMUL BIV5

```
#include <stdio.h>
main() /* citeste pe a si b, calculeaza si afiseaza radacina ecuatiei
                 ax + b = 0 */
{
 double a,b;
```

```
    if(scanf("%lf %lf", &a,&b) != 2)
        printf("coeficienti eronati\n");
    else
        if ( a != 0)
            printf("a=%g\tb=%g\tx=%g\n",a,b, -b/a);
        else
            if( b== 0)
                printf("ecuatie nedeterminata\n");
            else
                printf("ecuatia nu are solutie\n");
}
```

Observaţii:

1. Programul de faţă conţine o instrucţiune *if* imbricată.
2. Pentru a mări claritatea programelor se obişnuieşte să se decaleze spre dreapta (cu un tabulator) instrucţiunile din compunerea unei instrucţiuni *if*.
3. În acest program s-a realizat test relativ la valorile tastate pentru *a* şi *b*. Dacă funcţia *scanf* nu returnează valoarea 2, înseamnă că nu s-au tastat două numere la terminal. De aceea, într-o astfel de situaţie se afişează mesajul:

   "coeficienti eronati".

4.6 Să se scrie un program care citeşte coeficienţii *a*, *b*, *c*, *d*, *e*, *f*, ai unui sistem de două ecuaţii liniare cu două necunoscute, determină şi a-afişează soluţia acestuia cînd are o soluţie unică.

Fie sistemul de ecuaţii liniare:

ax+by=c

dx+ey=f

Notăm cu *det* determinantul coeficienţilor necunoscutelor. Acesta se calculează cu relaţia:

det=ae-bd

Notăm cu *det1* determinantul obţinut din *det* prin înlocuirea primei coloane cu coloana termenului liber şi cu *det2* determinantul obţinut din *det* prin înlocuirea coloanei a doua cu coloana termenului liber. Atunci:

det1=ce-bf şi det2=af-cd.

Procesul de calcul al valorilor lui *x* şi *y* se desfăşoară conform paşilor de

mai jos:

1. Se citesc valorile coeficienţilor

   a, b, c, d, e, f.

2. det=ae-bd.

3. Dacă *det*=0, atunci sistemul nu are soluţie unică (este nedeterminat sau incompatibil).

   Se afişează un mesaj corespunzător şi se întrerupe execuţia programului.

   Altfel se continuă cu punctul 4.

4. det1=ce-bf.

5. det2=af-cd.

6. x=det1/det.

7. y=det2/det.

8. Se afişează valorile lui *x* şi *y*.

PROGRAMUL BIV6

```
#include <stdio.h>
main() /* citeste pe a,b,c,d,e,f, determina si afiseaza solutia sistemului:
                ax + by = c
                dx + ey = f
             in cazul in care are solutie unica */
{
 double a,b,c,d,e,f,x,y,det,det1,det2;

 if(scanf("%lf %lf %lf %lf %lf %lf",&a,
       &b,&c,&d,&e,&f) !=6)
    printf("coeficienti eronati\n");
 else
    if((det = a*e - b*d) == 0)
       printf("sistemul are determinantul nul\n");
    else
    {
       det1 = c*e - b*f;
       det2 = a*f - c*d;
       x = det1/det;
       y = det2/det;
```

```
        printf("x=%g\ty=%g\n",x,y);
    }
}
```

Observaţii:

1. Expresia

   (1) (det=a*e-b*d)==0

   se evaluează astfel:

a. Se calculează

   a*e-b*d

   şi valoarea respectivă se atribuie lui *det*.

b. Se compară valoarea atribuită lui *det* cu zero.

   Dacă valoarea lui *det* este zero, atunci expresia (1) are valoarea *adevărat*; în caz contrar, expresia (1) are valoarea zero, adică *fals*.

2. Parantezele din expresia (1) sînt obligatorii.

   În lipsa lor, expresia (1) devine:

   (2) det=a*e-b*d==0.

   Aceasta se evaluează astfel:

a. Se calculează valoarea expresiei:

   a*e-b*d;

b. Deoarece operatorul == este mai prioritar decît =, se compară valoarea expresiei respective cu zero.

   Dacă expresia respectivă are valoarea zero, atunci expresia

   (3) a*e-b*d==0

   are valoarea 1 (*adevărat*),

   altfel are valoarea zero (*fals*).

c. Se atribuie lui *det* valoarea expresiei 3, adică 0 sau 1.

4.7 Să se scrie un program care citeşte valorile variabilelor *a, b, c*, calculează şi afişează rădăcinile ecuaţiei de gradul 2:

   a*x*x+b*x+c=0

Paşii procesului de calcul sînt:

1. Se citesc valorile variabilelor *a, b, c*.

2. Dacă a=b=c=0, ecuaţia este nedeterminată.

   Se afişează un mesaj corespunzător şi se întrerupe execuţia programului.

3. Dacă a=b=0 şi c este diferit de zero, ecuaţia nu are soluţie.

   Se afişează un mesaj corespunzător şi se întrerupe execuţia programului.

4. Dacă a=0 şi *b* este diferit de zero, ecuaţia se reduce la o ecuaţie de gradul întîi a cărui soluţie este:

   x=-c/b;

   se afişează un mesaj corespunzător, soluţia *x* şi se întrerupe execuţia programului.

5. Dacă *a* este diferit de zero, se calculează:

   delta=b*b-4*a*c

   şi

   d=2*a.

6. Dacă *delta* > 0, atunci delta=sqrt(delta); (*sqrt* este funcţia de bibliotecă prin care se calculează rădăcina pătrată).

   Se determină şi se afişează rădăcinile:

   x1=(-b+delta)/d;

   x2=(-b-delta)/d;

   apoi se întrerupe execuţia programului;

7. Dacă delta=0, atunci se determină şi se afişează rădăcina dublă:

   x=-b/d;

   se întrerupe execuţia programului.

8. Altfel, ecuaţia are rădăcini complexe conjugate.

   Se calculează:

   delta=sqrt(-delta).

   Se determină şi se afişează rădăcinile:

   x1=-b/d+i*delta/d;

   x2=-b/d-i*delta/d.

Se întrerupe execuţia programului.

PROGRAMUL BIV7

```
#include <stdio.h>
#include <math.h> // fisierul contine prototip pentru sqrt

main() /* citeste pe a,b,c, calculeaza si afiseaza radacinile ecuatiei:
          a*x*x + b*x + c = 0 */
{
 double a,b,c,d,delta;

 printf("coeficientul lui x patrat:");
 if(scanf("%lf",&a) != 1)
    printf("coeficientul lui x patrat eronat\n");
 else { /* 1*/
    printf("coeficientul lui x:");
    if(scanf("%lf",&b) != 1)
       printf("coeficientul lui x eronat\n");
    else {/* 2 */
       printf("termenul liber:");
       if(scanf("%lf", &c) != 1)
          printf("termenul liber eronat\n");
       else {/* 3 */
       /* afiseaza coeficientii cititi */
          printf("a=%g\tb=%g\tc=%g\n",a,b,c);
          if( a== 0 && b == 0 && c == 0)
               printf("ecuatie nedeterminata\n");
          else
             if( a == 0 && b == 0)
                printf("ecuatia nu are\
                        solutie\n");
             else
                if ( a == 0 ) {
                      printf("ecuatie de gradul\
                              1\n");
                      printf("x=%g\n",-c/b);
                }
                else{/* 4 *//* a != 0 */
                      delta = b*b - 4*a*c;
                      d = 2*a;
                      if( delta > 0 ) {
                         printf("ecuatia are 2\
```

```
                radacini reale si distincte\n");
                delta = sqrt(delta);
                printf("x1=%g\tx2=%g\n",
                  (-b + delta)/d,
                  (-b - delta)/d);
            }
            else
                if( delta == 0 ) {
                        printf("ecuatia are\
                              radacina dubla\n");
                        printf("x=%g\n", -b/d);
                }
                else {/* 5 */
                        printf("radacini\
                                complexe\n");
                        delta = sqrt(-delta)/d;
                        d = -b/d;
                        printf("x1=%g+i(%g)\n",d,
                               delta);
                        printf("x2=%g-i(%g)\n",d,
                               delta);
                }/* sfirsit*5**/
            }/*sfirsit *4**/
         }/* sfirsit*3**/
      }/*sfirsit *2**/
   }/*sfirsit*1**/
}/*sfirsit program*/
```

Observaţie:

Programul de faţă utilizează instrucţiunea *if* cu nivel de imbricare relativ mare. Programele care folosesc instrucţiuni *if* cu nivele mari de imbricare sînt greu de urmărit. Adesea se uită închiderea tuturor acoladelor. De aceea se recomandă reducerea pe cît posibil a nivelelor de imbricare ale instrucţiunilor *if*. În multe cazuri, nivelul de imbricare al unei instrucţiuni *if* se poate reduce folosind funcţia *exit* (vezi paragraful următor).

4.5. Funcţia standard exit

Funcţia *exit* are prototipul:

void exit(int *cod*)

şi el se află în fişierele de tip *h*:

stdlib.h

şi

process.h

La apelul acestei funcţii au loc următoarele acţiuni:

- se videază zonele tampon (bufferele) ale fişierelor deschise în scriere;
- se închid toate fişierele deschise;
- se întrerupe execuţia programului.

Parametrul acestei funcţii defineşte starea programului la momentul apelului.

Valoarea zero defineşte o terminare normală a execuţiei programului, iar o valoare diferită de zero semnalează prezenţa unei erori (terminarea anormală a execuţiei programului).

În concluzie, putem apela funcţia *exit* pentru a termina execuţia unui program, indiferent de faptul că acesta se termină normal sau din cauza unei erori.

Exerciţii:

4.8 Să se modifice programul din exerciţiul 4.7 în aşa fel încît el să nu mai conţină instrucţiuni *if* imbricate.

PROGRAMUL BIV8

```
#include <stdio.h>
#include <math.h>    // fisierul contine prototip pentru sqrt
#include <stdlib.h> // fisierul contine prototip pentru exit

main() /* citeste pe a,b,c, calculeaza si afiseaza radacinile ecuatiei:
               a*x*x + b*x + c = 0 */
{
 double a,b,c,d,delta;

 printf("coeficientul lui x patrat:");
 if(scanf("%lf",&a) != 1) {
    printf("coeficientul lui x patrat eronat\n");
    exit(1);/* terminare la eroare */
 }
 printf("coeficientul lui x:");
```

```
 if(scanf("%lf",&b) != 1) {
     printf("coeficientul lui x eronat\n");
     exit(1);/* terminare la eroare */
  }
  printf("termenul liber:");
  if(scanf("%lf", &c) != 1) {
     printf("termenul liber eronat\n");
     exit(1);/* terminare la eroare */
  }

/* afiseaza coeficientii cititi */
 printf("a=%g\tb=%g\tc=%g\n",a,b,c);
 if( a== 0 && b == 0 && c == 0) {
    printf("ecuatie nedeterminata\n");
    exit(0);/* terminare fara erori */
  }
 if( a == 0 && b == 0){
    printf("ecuatia nu are solutie\n");
    exit(0);
 }
 if ( a == 0 ) {
    printf("ecuatie de gradul 1\n");
    printf("x=%g\n",-c/b);
    exit(0);
 }
 delta = b*b - 4*a*c;
 d = 2*a;
 if( delta > 0 ) {
    printf("ecuatia are 2 radacini reale si\
           distincte\n");
    delta = sqrt(delta);
    printf("x1=%g\tx2=%g\n",(-b + delta)/d,
                             (-b - delta)/d);
    exit(0);
 }
 if( delta == 0 ) {
    printf("ecuatia are radacina dubla\n");
    printf("x=%g\n", -b/d);
    exit(0);
 }
 printf("radacini complexe\n");
 delta = sqrt(-delta)/d;
 d = -b/d;
```

```
   printf("x1=%g+i(%g)\n",d,delta);
   printf("x2=%g-i(%g)\n",d,delta);
}
```

4.6. Instrucţiunea while

Instrucţiunea *while* are formatul:

while(*expresie*)
instrucţiune

Primul rînd din acest format constituie antetul instrucţiunii *while*, iar *instrucţiune* este corpul ei.

La întîlnirea acestei instrucţiuni întîi se evaluează expresia din paranteze. Dacă ea are valoarea *adevărat* (este diferită de zero), atunci se execută *instrucţiune*. Apoi se revine la punctul în care se evaluează din nou valoarea expresiei din paranteze. În felul acesta, corpul ciclului se execută atît timp cît expresia din antetul ei este diferită de zero. În momentul în care *expresie* are valoarea zero, se trece la instrucţiunea următoare instrucţiunii *while*.

Corpul instrucţiunii *while* poate să nu se execute niciodată. Într-adevăr dacă *expresie* are valoarea zero de la început, atunci se trece la instrucţiunea următoare instrucţiunii *while* fără a executa niciodată corpul instrucţiunii respective.

Corpul instrucţiunii *while* este o singură instrucţiune, care poate fi compusă. În felul acesta, avem posibilitatea să executăm repetat mai multe instrucţiuni grupate într-o instrucţiune compusă.

Corpul instrucţiunii *while* poate fi o altă instrucţiune *while* sau să fie o instrucţiune compusă care să conţină instrucţiuni *while*. În acest caz se spune că instrucţiunile *while* respective sînt imbricate.

Menţionăm că instrucţiunea din corpul unei instrucţiuni *while* poate să definească un alt mod de execuţie a instrucţiunii *while* decît cel indicat mai sus. Astfel, ea poate realiza terminarea instrucţiunii *while* fără a se mai ajunge la evaluarea expresiei din antetul ei. De exemplu, dacă în corpul unei instrucţiuni *while* se apelează funcţia *exit*, atunci se va termina execuţia ciclului *while*, deoarece se întrerupe chiar execuţia programului.

Despre instrucţiunea *while* se spune adesea că este o instrucţiune *ciclică*.

Amintim că ea defineşte o structură repetitivă condiţionată anterior.

Exerciţii:

4.9 Să se scrie un program care calculează şi afişează valoarea polinomului

```
p(x)=3*x*x-7*x-10
```

pentru

```
x=1, 2, ..., 10
```

Programul de faţă realizează un proces de calcul care constă din evaluarea şi afişarea valorii expresiei:

```
3*x*x-7*x-10
```

pentru cele 10 valori ale lui *x*. Pentru o valoare a lui *x*, acest lucru se realizează prin intermediul instrucţiunii de apel:

```
printf("x=%d\tp(x)=%d\n",x,3*x*x-7*x-10);
```

Rezultă că, această instrucţiune trebuie să se execute repetat pentru x=1, 2, ..., 10. În acest scop, inţial vom atribui lui *x* valoarea 1:

```
x=1;
```

Apoi, vom utiliza o instrucţiune *while* care să permită execuţia repetată a instrucţiunii de apel a funcţiei *printf* de mai sus.

Deoarece lui *x* i s-a atribuit deja valoarea 1, rezultă că la prima execuţie a acestei instrucţiuni se calculează *p(1)*. Pentru ca la a doua execuţie a ei să se calculeze *p(2)*, este necesar să se mărească valoarea lui *x* cu 1. Aceasta se poate realiza folosind instrucţiunea expresie

```
x++;
```

Cu alte cuvinte, este suficient să se execute repetat secvenţa de instrucţiuni:

```
printf("x=%d\tp(x)=%d\n",x,3*x*x-7*x-10);
x++;
```

x avînd iniţial valoarea 1. Această secvenţă formează corpul ciclului. De aceea, ea va fi inclusă între acolade pentru a forma o singură instrucţiune.

Deoarece ultima execuţie a secvenţei de mai sus are loc cînd x=10, rezultă că ea se va executa repetat atîta timp cît x <= 10. De aceea, antetul ciclului *while* va fi:

```
while(x<=10)
```

În concluzie, procesul de calcul ce urmează a fi realizat îl putem descrie astfel:

1. x=1
2. Atît timp cît x<=10 se execută:

2.1. Afişează pe x şi $p(x)$.

2.2. Incrementează pe x.

PROGRAMUL BIV9

```
#include <stdio.h>
main() /* afiseaza valorile polinomului
                p(x)=3*x*x - 7*x - 10 pentru x = 1,2,...,10 */
{
 int x;

 x = 1;
 while( x <= 10) {
     printf("x=%d\tp(x)=%d\n", x,3*x*x - 7*x -10);
     x++;
 }/* sfirsit while */
}/* sfirsit main*/
```

Observaţie:

Limbajul C permite scrierea de programe compacte. De exemplu, secvenţa de mai sus poate fi compactată înlocuind expresia

x <= 10

cu

++x <= 10

În felul acesta se elimină instrucţiunea expresie

x++;

din corpul ciclului.

Ţinînd seama de ordinea operaţiilor, expresia

++x <= 10

se evaluează astfel:

1. Se incrementează x.
2. Se compară valoarea incrementată a lui x cu 10.

   Dacă aceasta nu-l depăşeşte pe 10, atunci se execută corpul ciclului, care acum constă numai din apelul funcţiei *printf*.

   În caz contrar se încheie execuţia ciclului *while* şi prin aceasta se termină

şi execuţia programului.

În acest caz se va schimba atribuirea

x=1

cu

x=0

deoarece execuţia corpului ciclului începe după incrementarea lui *x*. Cu alte cuvinte, programul BIV9 poate fi rescris astfel:

```
...
 x=0;
 while(++x<=10)
     printf("x=%d\t p(x)=%d\n",x,3*x*x-7*x-10);
} /* sfirsit main */
```

Menţionăm că expresia

x++ <= 10

utilizată în antetul ciclului nu realizează acelaşi lucru.

Într-adevăr, în cazul expresiei

++x <= 10

atît comparaţia, cît şi corpul ciclului *while* se execută cu valoarea incrementată a lui *x*.

În cazul expresiei

x++ <= 10

comparaţia se face cu valoarea neincrementată a lui *x*, iar corpul ciclului *while* se execută cu valoarea incrementată a lui *x*. De aceea, dacă x=10, atunci

++x <= 10

este falsă, deci corpul ciclului nu se mai execută, în schimb

x++ <= 10

are valoarea adevărat şi se va executa corpul ciclului şi pentru x=11. De aceea, se va obţine acelaşi lucru dacă în antetul ciclului vom folosi expresia

x++<10

4.10 Să se scrie un program care tabelează valorile funcţiei *sin(x)* cu pasul

de un grad sexagesimal, *x* aparţinînd intervalului [0,360).

Funcţia *sinus*, poate fi apelată ca funcţie standard şi are prototipul

double sin(double *x*);

în fişierul *math.h*.

Parametrul *x* este în radiani.

Dacă *a* este un unghi în grade sexagesimale, atunci el se reprezintă în radiani dacă se înmulţeşte cu factorul PI/180 (PI=3.14159265...):

a*PI/180

Tabelarea funcţiei *sinus* se va realiza conform paşilor de mai jos:

1. f=PI/180
2. x=0
3. Atît timp cît x<=359, se execută:

3.1. Calculează şi afişează sin(x*f).

3.2. Incrementează pe *x*.

Pasul 3 de mai sus este analog cu pasul 2 din exemplul precedent. De aceea, el se realizează printr-un ciclu *while* asemănător. La realizarea lui vom face compactarea indicată în observaţia de la exemplul respectiv.

PROGRAMUL BIV10

```
#include <stdio.h>
#include <math.h>

#define PI 3.14159265358979

main() /* calculeaza si afiseaza valorile functiei sin(x) din grad
          in grad sexagesimal, x apartinind intervalului [0,359] */
{
 int x;
 double f;

 f = PI/180.0;
 x = -1;
 while(++x <= 359 )
    printf("sin(%d)=%.16f\n", x,sin(x*f));
}
```

Observaţie:

Valorile funcţiei *sinus* se afişează cîte una pe o linie. De aceea, programul afişează 360 de linii. Utilizatorul are posibilitatea de a vedea pe ecran, în fereastra utilizator (tastînd <Alt>-F5), numai ultimele 25 de valori. Aceasta din cauză că un ecran, în mod text, se compune din 25 de linii a 80 de coloane.

În astfel de situaţii se pune problema blocării execuţiei unui program după ce acesta a afişat un ecran de rezultate. După analiza datelor de pe ecran, utilizatorul urmează să deblocheze execuţia programului acţionînd o tastă oarecare.

Un mod simplu de a bloca execuţia unui program este acela de a apela funcţia *getch* în momentul în care se doreşte să se analizeze datele afişate în fereastra utilizator.

La apelul funcţiei *getch* programul se blochează deoarece se aşteaptă acţionarea unei taste. Totodată se afişează automat fereastra utilizator. La acţionarea unei taste oarecare, programul se deblochează.

În exemplul de faţă vom apela funcţia *getch* ori de cîte ori programul a afişat 23 de valori ale funcţiei sinus. Aceste valori ocupă 23 de linii ale ecranului. Pe linia 24 se va afişa textul:

pentru a continua actionati o tasta

Funcţia *getch* se apelează în corpul lui *while* de îndată ce sînt afişate 23 de valori. În acest scop utilizăm valorile lui *x* care cresc de la 0 la 359. Ecranul trebuie blocat după apelul funcţiei *printf* şi în momentul în care x+1 are valorile:

23, 46, 69,...

adică atunci cînd x+1 este multiplu de 23.

Aceasta se exprimă prin expresia:

(x+1)%23==0

Deci funcţia *getch* se apelează cînd expresia de mai sus este *adevărată*.Înainte de a apela funcţia *getch* va trebui apelată funcţia *printf* pentru a afişa, pe linia 24, textul:

pentru a continua actionati o tasta

Ţinînd seama de aceste observaţii, modificăm programul de faţă ca mai jos.

PROGRAMUL BIV10A

```
#include <stdio.h>
#include <math.h>
#include <conio.h>

#define PI 3.14159265358979

main() /* calculeaza si afiseaza valorile functiei sin(x) din grad
          in grad sexagesimal, x apartinind intervalului [0,359] */
{
 int x;
 double f;

 f = PI/180.0;
 x = -1;
 while(++x <= 359 ) {
     printf("sin(%d)=%.16f\n", x,sin(x*f));
     if((x+1)%23 == 0 ) {
         printf("pentru a continua actionati o\
                 tasta\n");
         getch();
     }
 }/* sfirsit while */
} /* sfirsit main*/
```

4.11 Să se scrie un program care citeşte un şir de întregi separaţi prin caractere albe şi afişează suma lor. După ultimul număr se va tasta un caracter alb urmat de un caracter nenumeric (de exemplu sfîrşitul de fişier: <Ctrl>-z, o literă etc), iar după acesta se va acţiona tasta *Enter*.

Programul de faţă are o parte care se execută repetat:

a. Citeşte construcţia curentă de la intrare.

b. Dacă la intrare s-a aflat un întreg, atunci acesta se adună la valoarea unei variabile *s* şi se revine la punctul a.

Altfel se întrerupe citirea.

De aici rezultă că iniţial s trebuie să aibă valoarea zero.

Un întreg se citeşte apelînd funcţia *scanf*. Dacă notăm cu *i* variabila căreia i se atribuie întregul citit, atunci la citirea unui întreg, expresia:

scanf("%d",&i)==1

are valoarea *adevărat*. Deci valoarea lui *i* trebuie adunată la *s* atît timp cît

expresia de mai sus este *adevărată*. Deci expresia de mai sus se utilizează în antetul instrucţiunii *while*, iar corpul ciclului se compune din instrucţiunea:

s=s+i;

PROGRAMUL BIV11

```
#include <stdio.h>
main() /* citeste un sir de intregi separati prin caractere
        albe si afiseaza suma lor */
{
 int i,s;

 s = 0;
 while(scanf("%d",&i) == 1)
     s += i;
 printf("suma = %d\n", s);
}
```

4.12 Să se scrie un program care citeşte un întreg *n*, calculează şi afişează pe *n!*.

Avem:

n!=1*2*3*...*(n-1)*n, pentru n>0 şi 0! se consideră egal cu 1.

Acest calcul implică un proces ciclic.

Într-adevăr, fie

f=1 şi i=2

Atunci:

f*i=1*2=2!

Considerăm instrucţiunea de atribuire:

f=f*i;

Cum f*i=2!, rezultă că f=2!. Putem folosi aceeaşi instrucţiune pentru a calcula pe 3!.

Într-adevăr

3!=2!*3=f*3

deci expresia: f*i=3! dacă, în prealabil, *i* se măreşte cu o unitate. Rezultă că alături de instrucţiunea de mai sus, este necesar să considerăm şi instrucţiunea de incrementare a lui *i*:

a. f=f*i;

b. i=i+1;

Această secvenţă se execută repetat, întîi *a* apoi *b* şi apoi din nou *a* şi apoi *b* şi aşa mai departe. Lui *f* i se atribuie succesiv valorile:

2!, 3!, 4!,...

Pentru a calcula pe n! (n>1), secvenţa de instrucţiuni a-b trebuie să se execute repetat atît timp cît *i* nu-l depăşeşte pe *n*, adică atît timp cît expresia:

i<=n

este *adevărată*.

Acest proces de calcul se realizează simplu prin instrucţiunea *while* de mai jos:

```
while(i<=n){
 f=f*i;
 i++;
}
```

Instrucţiunea compusă poate fi compactată astfel:

```
f=f*i++;
```

Într-adevăr, operatorul de incrementare se aplică postfixat, deci *f* se înmulţeşte cu valoarea neincrementată a lui *i*, ca şi în cazul instrucţiunii compuse iniţiale.

PROGRAMUL BIV12

```
#include <stdio.h>
#include <stdlib.h>

main() /* citeste pe n din intervalul [1,170], calculeaza
          si afiseaza pe n! */
{
 int n,i;
 double f;

 printf("valoarea lui n:");
 if(scanf("%d",&n) != 1 ) {
    printf("nu s-a tastat un intreg\n");
    exit(1);
 }
 if( n < 0 || n > 170 ) {
```

```
        printf("n nu apartine intervalului\
                [0,170]\n");
        exit(1);
    }
    f = 1.0;
    i = 2;
    while( i <= n )
        f = f*i++;
    printf("n=%d\tn!=%g\n",n,f);
}
```

4.13 Să se scrie un program care citeşte componentele a doi vectori x şi y, calculează şi afişează valoarea produsului lor scalar.

Dacă vectorul x are componentele:

x1, x2, ..., xn

iar y:

y1, y2, ..., yn

atunci produsul lor scalar se defineşte prin suma de produse:

(x,y)=x1*y1+x2*y2+...+xn*yn

Componentele celor doi vectori se tastează în următoarea ordine:

x1 y1

x2 y2

...

xn yn

Pe rîndul care urmează după ultima pereche se poate tasta o literă, un caracter special care nu intră în compunerea unui număr sau sfîrşitul de fişier.

PROGRAMUL BIV13

```
#include <stdio.h>
main() /* citeste componentele vectorilor x si y, calculeaza si
          afiseaza valoarea produsului lor scalar */
{
 double x,y,prodscal;
 int i;
```

```
    prodscal = 0.0;
    printf("componentele lui x si y\n");
    i = 0;
    printf("x[%d]\ty[%d]:", i+1,i+1);
    while(scanf("%lf %lf", &x,&y) == 2) {
        prodscal += x*y;
        i++;
        printf("x[%d]\ty[%d]:",i+1,i+1);
    }
    printf("numarul componentelor vectorilor=%d\n",i);
    printf("produsul scalar=%g\n",prodscal);
}
```

4.14 Să se scrie un program care citeşte fără ecou caractere imprimabile şi le afişează ca şi caractere minus. Programul se întrerupe la tastarea unui caracter care nu aparţine codului ASCII.

La realizarea programului vom folosi funcţiile *getch* şi *putch*.

Funcţia *getch* returnează codul ASCII al caracterului citit sau zero dacă se citeşte un caracter care nu aparţine codului ASCII. Expresia

c=getch()

atribuie variabilei *c* codul ASCII al caracterului citit.Se vor recodifica prin minus numai caracterele imprimabile, adică de cod ASCII cel puţin egal cu 32 şi cel mult 126. Restul caracterelor se neglijează. Dacă c are ca valoare codul ASCII al caracterului citit, atunci instrucţiunea

```
if(c>=32&&c<=126)
 putch('-');
```

permite afişarea caracterului respectiv ca şi caracter minus, dacă acesta este un caracter imprimabil; altfel el este neglijat. Această instrucţiune se execută repetat, cît timp se tastează un caracter care aparţine codului ASCII. De aceea, ea va forma corpul unui ciclu *while*.

În antetul acestui ciclu vom folosi expresia

c=getch()

care este diferită de zero (adevărată) pentru caracterele codului ASCII şi zero (falsă) pentru restul caracterelor.

PROGRAMUL BIV14

```
#include <conio.h>
main() /* citeste caractere imprimabile fara ecou si le afiseaza
```

```
        ca si caracter minus */
{
 int c;

 while ( c = getch() )
     if( c >= 32 && c <= 126 )
         putch('-');
}
```

4.15 Să se scrie un program care apelează funcţia *putch* cu valorile codului ASCII extins.

Codul ASCII extins cuprinde valorile întregi din intervalul [0,255].

Pe o linie se va afişa valoarea codului urmată de imaginea corespunzătoare afişată prin funcţia *putch*. Ecranul se blochează după 23 de linii afişate sub forma indicată mai sus şi după ce pe linia 24 se afişează mesajul:

pentru a continua actionati o tasta

După acţionarea unei taste ecranul se deblochează pentru a se afişa un nou set de imagini şi aşa mai departe.

PROGRAMUL BIV15

```
#include <stdio.h>
#include <conio.h>

main() /* apeleaza functia putch cu valorile codului ASCII extins */
{
 int c;

 c = -1;
 while ( ++c <= 255 ) {
     printf("cod:%4d\t",c);
     putch(c);
     putchar('\n');
     if(( c+1) % 22 == 0) {
         printf("pentru a continua actionati o\
                  tasta\n");
         getch();
     }
 }
}
```

4.16 Să se scrie un program care citeşte rezultatele unor măsurători:

x1, x2, ..., xn

calculează şi afişează:

- media aritmetică a măsurătorilor;
- media geometrică a lor;

şi

- abaterea pătratică.

Dacă notăm cu *meda* media aritmetică a măsurătorilor, atunci abaterea pătratică este rădăcina pătrată din expresia:

[(x1-meda)**2+(x2-meda)**2+...+(xn-meda)**2]/N

unde:

N - Este n sau n-1.

Numitorul se ia egal cu n-1 cînd *n* este relativ mic. De obicei:

N=n-1 pentru n < 30

şi

N=n pentru n >= 30

Evident n >= 2.

Deoarece pentru calculul abaterii pătratice este nevoie de măsurătorile x1, x2, ..., xn, cît şi de media lor aritmetică, programul va trebui să păstreze măsurătorile respective într-un tablou unidimensional, pe care îl notăm cu *x*.

Vom presupune că există cel mult 500 de măsurători (n<=500).

Media geometrică se calculează prin extragerea rădăcinii de ordinul ng din produsul celor ng măsurători pozitive dintre cele n măsurători citite.

Setul de măsurători sînt date flotante în simplă precizie. Setul de măsurători se consideră terminat dacă se tastează un caracter care nu intră în compunerea unei date numerice.

Programul listează măsurătorile citite, cîte 5 pe un rînd. De asemenea, programul se blochează după afişarea a 23 de linii cu măsurători.

PROGRAMUL BIV16

```
#define MAXMAS 500
```

```
#define LIMNUMIT 30
#define APEL printf("masuratoarea a %d-a:", n+1)

#include <stdio.h>
#include <math.h>
#include <conio.h>
#include <stdlib.h>

main() /* citeste un set de masuratori, calculeaza si afiseaza
          media lor aritmetica, geometrica si abaterea patratica */
{
 float x[MAXMAS];
 double meda,medg,ap;
 int n,ng,i;

 n=ng=0;
 meda=0.0;
 medg=1.0;
 APEL;

/* - se citesc masuratorile;
   - se insumeaza;
   - se inmultesc cele pozitive
*/
 while(scanf("%f", &x[n] ) == 1) {
     meda += x[n];
     if( x[n] > 0 ) {
         medg *= x[n];
         ng++;
     }
 n++;
 APEL;
 }
 if( n < 1 ) {
     printf("nu s-a tastat nici o masuratoare\n");
     exit(1);
 }
 meda /= n;

/* calculeaza media geometrica */
 if( ng < 2 ) {
     printf("nu exista nici 2 masuratori\
            pozitive\n");
```

```
        medg = -1.0;
    }
    else
        medg = pow(medg,1.0/ng);
    if( n > 1) {
/* calculeaza abaterea patratica */
        ap = 0.0;
        i = 0;
        while ( i < n ) {
            ap += (x[i] - meda)*(x[i] - meda);
            i++;
        }
        if( n < LIMNUMIT)
            ap /= n-1;
        else
            ap /= n;
    }
    else {
        printf("nu exista nici 2 masuratori\n");
        ap = -1;
    }
/* listeaza masuratorile, cite 5 pe un rind */
    i = 0;
    while(i < n) {
        printf("x[%d]=%g%c",i+1,x[i],
                    i%5==4 || i==n-1 ? '\n':' ');
        i++;
        if(i%(5*23) == 0 ) {
            printf("pentru a continua actionati o
                    tasta\n");
            getch();
        }
    }
/* afiseaza mediile si abaterea patratica */
    printf("media aritmetica=%g\n", meda);
    if(medg != -1.0)
        printf("media geometrica=%g\n",medg);
    if( n > 1)
        printf("abaterea patratica=%g\n",sqrt(ap));
}
```

Observaţii:

1. Pentru a afişa măsurătorile cîte 5 pe un rînd s-a apelat funcţia *printf* folosind trei specificatori de format în parametrul de *control*:

   "x[%d] = %g%c"

Parametrul corespunzător lui %d este i+1. Acesta defineşte numărul de ordine al măsurătorii curente.

Parametrul corespunzător lui *%g* este x[i] şi reprezintă chiar valoarea măsurătorii curente.

După specificatorul *%g* urmează specificatorul *%c*, a cărui parametru defineşte caracterul care se afişează după valoarea măsurătorii curente. Acest caracter poate fi spaţiu (' ') sau caracterul de rînd nou ('\n').

Pe primul rînd se afişează valorile:

x[0] x[1] x[2] x[3] x[4]

deci, după măsurătorile de indice 0, 1, 2 şi 3 se afişează un spaţiu, iar după x[4] urmează caracterul de rînd nou.

Analog, pe rîndul al doilea se afişează valorile:

x[5] x[6] x[7] x[8] x[9]

deci, după măsurătorile de indice 5, 6, 7 şi 8 se afişează un spaţiu, iar după x[9] urmează caracterul de rînd nou.

În general, pe un rînd se afişează valorile.

x[5k] x[5k+1] x[5k+2] x[5k+3] x[5k+4]

Se observă că după o măsurătoare de indice 5k+4 urmează caracterul de rînd nou, iar în rest urmează un spaţiu.

Un indice *i* este de forma 5k+4, dacă restul împărţirii lui *i* la 5 este egal cu 4, adică, dacă expresia:

i%5==4

este adevărată.

De asemenea, după ultima măsurătoare se va afişa caracterul de rînd nou, adică dacă

i==n-1

are valoarea *adevărat*.

Rezultă că afişarea caracterului de rînd nou se va face atunci cînd expresia

```
i%5==4||i==n-1
```

este *adevărată*. În rest se afişează un caracter spaţiu. Această situaţie se reprezintă simplu prin expresia condiţională:

```
i%5==4||i==n-1?'\n':' '
```

expresie ce corespunde specificatorului de format %c.

Acest exemplu demonstrează odată în plus facilităţile mari oferite de expresiile condiţionale la compactarea programelor.

Această metodă de afişare a fost indicată de autorii limbajului C în lucrarea [2].

2. Pentru a calcula radicalul de ordinul *ng* din *medg* se apelează funcţia *pow* cu parametrul *medg* şi 1.0/ng.

   Funcţia *pow* are prototipul:

   double pow (double *x*,**double** *y*);

şi returnează pe *x* la puterea *y*. În cazul de faţă 1.0/ng are tipul *double* deoarece 1.0 este o constantă de tip *double*. Menţionăm că expresia 1/ng are valoarea zero (împărţire de întregi cu ng>1) deci este necesară utilizarea constantei 1.0. Apelul funcţiei *pow* în programul de mai sus returnează pe *medg* la puterea 1.0/ng, adică radical indice *ng* din *medg*.

4.17 Să se scrie un program care citeşte un cuvînt (succesiune de caractere diferite de caracterele albe) şi afişează prefixele şi sufixele lui diferite de el însuşi.

Fie cuvîntul:

```
e+t*f
```

atunci prefixele acestui cuvînt, diferite de el însuşi sînt:

```
e
e+
e+t
e+t*
```

În mod analog, sufixele aceluiaşi cuvînt sînt:

```
f
*f
t*f
+t*f
```

Se presupune că un cuvînt nu depăşeşte 70 de caractere.

Programul avansează peste eventualele caractere albe care preced cuvîntul de prelucrat. Apoi citeşte caracterele din compunerea cuvîntului şi le păstrează ca elemente ale tabloului *tab*.

Cuvîntul se termină la:

- întîlnirea unui caracter alb;
- întîlnirea sfîrşitului de fişier;
- după ce s-au citit 70 de caractere care nu sînt albe.

Caracterele se citesc folosind macroul *getchar*. Acesta se apelează prin expresia de atribuire:

c=getchar()

care atribuie lui *c* codul ASCII al caracterului citit.Pentru a avansa peste caracterele albe este necesar să stabilim dacă codul atribuit lui *c* este codul spaţiului, al tabulatorului sau al caracterului de rînd nou. Expresia

(c=getchar())==' '

compară codul caracterului citit cu cel al spaţiului şi are valoarea *adevărat*, dacă s-a citit un spaţiu şi *fals* în caz contrar.

În mod analog, expresia

c=='\t'

este adevărată dacă s-a citit caracterul tabulator, iar expresia

c=='\n'

este adevărată dacă s-a citit caracterul *newline*. Rezultă că, dacă se citeşte un caracter alb, atunci este adevărată una dintre cele trei expresii de mai sus, ceea ce conduce la faptul că expresia:

(1) (c=getchar())==' '||c=='\t'||c=='\n'

este adevărată dacă şi numai dacă se citeşte un caracter alb.

De aici, rezultă că pentru a avansa peste caracterele albe vom utiliza un ciclu *while* care se va executa atît timp cît expresia (1) este adevărată. Deci antetul ciclului este:

while((c=getchar())==' '||c=='\t'||c=='\n')

Deoarece acest ciclu nu are de realizat altceva decît să avanseze peste caracterele albe, corpul lui se compune din instrucţiunea vidă.

Citirea caracterelor cuvîntului de la intrare se realizează tot printr-o

instrucţiune *while*. De data aceasta, ciclul *while* se execută atîta timp cît expresia (1) este falsă, nu s-a întîlnit sfîrşitul de fişier şi nici nu s-au citit încă 70 de caractere.

Prima condiţie se exprimă prin negaţia expresiei (1), presupunînd că *c* conţine deja caracterul curent.

(2) !(c==' '||c=='\t'||c=='\n')

Ţinînd seama de faptul că expresia

!(a||b)

este echivalentă cu:

!a&&!b

expresia (2) se poate pune sub forma:

(3) !(c==' ')&&!(c=='\t')&&!(c=='\n')

Deoarece expresia:

!(a==b)

este echivalentă cu:

a!=b

expresia (3) se poate pune sub o formă mai simplă:

(4) c!=' '&&c!='\t'&&c!='\n'

A doua condiţie pentru continuarea ciclului *while* este ca să nu se citească sfîrşitul de fişier. Aceasta înseamnă că expresia:

(5) c!=EOF

trebuie să fie adevărată.

Dacă notăm cu *i* numărătorul de caractere citite, diferite de caracterele albe, atunci ciclul *while* continuă dacă pe lîngă condiţiile (4) şi (5) de mai sus, este adevărată şi condiţia exprimată prin expresia:

(6) i<70 (se presupune că iniţial *i* a fost 0).

De aici rezultă că ciclul *while* pentru citirea caracterelor cuvîntului aflat la intrare se continuă atît timp cît expresiile (4), (5) şi (6) sînt adevărate, ceea ce se exprimă conectînd expresiile respective prin operatorul "şi logic".

În corpul ciclului se păstrează codul caracterului citit în tabloul *t* de tip caracter, se incrementează numărătorul de caractere şi apoi se citeşte caracterul următor. Acestea se realizează prin instrucţiunile de atribuire:

t[i++]=c;

şi

c=getchar();

Pentru determinarea prefixelor cuvîntului urmăm paşii de mai jos

unde:

i	- Are ca valoare lungimea cuvîntului.
j	- Are ca valoare lungimea prefixului curent.
k	- Numără caracterele unui prefix.

1. j=1 - primul prefix are lungimea 1.

2. Cît timp j<i se execută:

2.1. k=0.

2.2. Cît timp k<j se execută:

2.2.1. Se afişează t[k].

2.2.2. Se măreşte k cu o unitate.

2.3. Se afişează *newline* deoarece s-au afişat toate caracterele prefixului curent.

2.4. Se măreşte *j* cu o unitate pentru a afişa prefixul următor.

Pasul 2 se realizează prin două cicluri *while* imbricate:

```
while(j<i){
 k=0; /* punctul 2.1. */
 while(k<j){ /* punctul 2.2. */
     putchar(t[k]); /* punctul 2.2.1. */
     k++; /* punctul 2.2.2. */
 }
 putchar('\n'); /* punctul 2.3. */
 j++; /* punctul 2.4. */
}
```

Sufixele se pot determina printr-o secvenţă analogă. Dacă în cazul prefixelor acestea toate încep cu t[0], în cazul sufixelor, caracterul de început este variabil. Astfel, sufixul de lungime 1 conţine numai ultimul caracter al cuvîntului, deci este t[i-1]. Sufixul de lungime 2 se compune din ultimele 2 caractere ale cuvîntului:

t[i-2] şi t[i-1].

Sufixul de lungime 3 se compune din caracterele:

t[i-3], t[i-2] şi t[i-1].

În general, sufixul de lungime *j* se compune din caracterele:

t[i-j], t[i-j+1], ..., t[i-1].

De aceea, în acest caz variabila *k* nu se mai iniţializează cu 0, ci cu i-j.

Deoarece toate sufixele au ca ultim caracter pe t[i-1], rezultă că variabila *k* variază cu pasul 1 de la i-j pînă la i-1. În rest, secvenţa pentru determinarea sufixelor este aceeaşi cu cea pentru determinarea prefixelor.

PROGRAMUL BIV17

```
#include <stdio.h>

#define MAX 70

#include <stdlib.h>

main() /* citeste un cuvint si afiseaza prefixele si sufixele lui */
{
 int c,i,j,k;
 char t[MAX];

/* avans peste eventualele caractere albe care preced cuvintul
  de prelucrat */
 while((c = getchar() ) == ' '|| c== '\t'||
        c == '\n')
     ;
 if( c == EOF ) {
     printf("cuvint vid\n");
     exit(1);
 }

/* se citesc caracterele cuvintului de la intrare */
 i = 0;/* numarator de caractere*/
 while(c !=' ' && c != '\t' && c != '\n'
       && c != EOF && i < 70) {
     t[i++] = c;
     c = getchar();
 }

/* afiseaza prefixele cuvintului de lungime i */
```

```
    j = 1; /* j are ca valoare lungimea prefixului curent */
    while( j < i ) {
        k=0;/* numarator pentru caracterele prefixului: de la 0 la j-1 */
        while( k < j )
            putchar(t[k++]);
        putchar('\n'); /* dupa prefixul afisat */
        j++;
    } /* sfirsit afisare prefixe */

   /* afiseaza sufixele cuvintului citit */
    j = 1; /* lungimea sufixului curent*/
    while( j < i ) {
        k = i-j; /* numarator pentru caracterele sufixului:
                                de la i-j la i-1 */
        while( k < i )
            putchar(t[k++]);
        putchar('\n');
        j++;
    }
   }
```

4.7. Instrucţiunea for

Instrucţiunea *for*, ca şi instrucţiunea *while*, se utilizează pentru a realiza o structură *repetitivă condiţionată anterior*.

Formatul ei este:

for(*exp1;exp2;exp3*)
instrucţiune

unde:

exp1, exp2 şi exp3 - Sînt expresii.

Antetul instrucţiunii *for* este:

for(exp1;exp2;exp3)

iar *instrucţiune* este corpul ei şi ea se execută repetat.

Expresia *exp1* se numeşte partea de *iniţializare* a ciclului *for*, iar *exp3* este partea de *reiniţializare* a lui. Expresia *exp2* este condiţia de continuare a ciclului *for* şi ea joacă acelaşi rol cu expresia din ciclul *while*.

Instrucţiunea *for* se execută astfel:

1. Se execută secvenţa de iniţializare definită de *exp1*.
2. Se evaluează expresia *exp2*.

   Dacă are o valoare diferită de zero (este *adevărată*), atunci se execută instrucţiunea care formează corpul ciclului.

   Altfel (expresia are valoarea zero adică *fals*) se termină execuţia instrucţiunii *for* şi se trece la instrucţiunea următoare.
3. După executarea corpului ciclului se execută secvenţa de reiniţializare definită de *exp3*.

   Apoi se reia execuţia de la pasul 2.

Ca şi în cazul instrucţiunii *while*, instrucţiunea din corpul ciclului *for* nu se execută niciodată dacă *exp2* are valoarea zero chiar de la început.

Expresiile din antetul lui *for* pot fi şi vide. Caracterele punct şi virgulă vor fi totdeauna prezente.

Exemplu:

Secvenţa de însumare a elementelor tabloului *tab*:

s=tab[0]+tab[1]+...+tab[n-1]

se realizează executînd repetat instrucţiunea compusă:

```
{
 s=s+tab[i];
 i++;
}
```

unde iniţial s=0 şi i=0, atît timp cît i<n. Ea se poate realiza cu ajutorul instrucţiunii *for* de mai jos:

```
for(s=0,i=0;i<n;i++)
 s=s+tab[i];
```

În acest caz expresia *exp1* este formată din două atribuiri:

s=0, i=0

exp2 este reprezentată de condiţia

i<n

iar *exp3* este expresia

i++

Conform definiţiei instrucţiunii *for*, la început se evaluează *exp1*, deci se execută atribuirile

s=0

şi

i=0

Se evaluează *exp2*, adică se determină valoarea condiţiei

i<n

Dacă ea este adevărată, atunci se execută corpul instrucţiunii *for*, adică suma

s=s+tab[i]

Apoi se evaluează *exp3*, adică se execută incrementarea

i++;

După incrementarea lui *i* se revine la evaluarea condiţiei *exp2*. În felul acesta, ciclul continuă pînă cînd condiţia i<n devine falsă, adică în momentul în care *i* devine egal cu *n*. Deci ciclul se întrerupe după ce elementele:

tab[0], tab[1], ..., tab[n-1]

au fost adăugate la *s*.

Secvenţa de mai sus poate fi scrisă cu ajutorul instrucţiunii *while* astfel:

```
s=0;
i=0;
while(i<n){
 s=s+tab[i];
 i++;
}
```

În general, instrucţiunea *for*:

for(*exp1;exp2;exp3*)
instrucţiune

poate fi scrisă cu ajutorul unei secvenţe în care se utilizează instrucţiunea *while*:

exp1;
while(*exp2*)
{
instrucţiune

exp3;
}

Această echivalare nu are loc într-un singur caz şi anume atunci cînd, în corpul instrucţiunii se utilizează instrucţiunea *continue* (vezi mai jos).

Reciproc, orice instrucţiune *while* poate fi scrisă cu ajutorul unei instrucţiuni *for* în care *exp1* şi *exp3* sînt vide.

Astfel, instrucţiunea *while* de mai jos:

while(*exp*)
instrucţiune

este echivalentă cu instrucţiunea *for*

for(;*exp*;)
instrucţiune

O instrucţiune *for* de forma

for(;;)
instrucţiune

este validă şi ea este echivalentă cu instrucţiunea *while* de mai jos:

while(1)
instrucţiune

Un astfel de ciclu se poate termina prin alte mijloace decît cel obişnuit, cum ar fi instrucţiunea de revenire dintr-o funcţie, un salt la o etichetă etc. (vezi mai jos).

Din cele de mai sus rezultă echivalenţa celor două cicluri. Autorii recomandă utilizarea instrucţiunii *for* în ciclurile în care sînt prezente părţile de iniţializare şi reiniţializare. De obicei, aceste cicluri sînt aşa numitele cicluri cu *pas*.

Exerciţii:

4.18 Să se scrie un program care citeşte întregul n din intervalul [0,170], calculează şi afişează pe n! .

Acest exerciţiu a fost rezolvat cu ajutorul instrucţiunii *while*. Propunem cititorului să rescrie exerciţiile rezolvate cu ajutorul instrucţiunii *while* înlocuind-o pe aceasta cu *for*.

PROGRAMUL BIV18

```
#include <stdio.h>
#include <stdlib.h>
```

```
main() /* citeste pe n din intervalul [0,170], calculeaza
          si afiseaza pe n! */
{
 int n,i;
 double f;

 printf("valoarea lui n:");
 if(scanf("%d",&n) != 1) {
    printf("nu s-a tastat un intreg\n");
    exit(1);
 }
 if( n < 0 || n > 170 ) {
    printf("n nu apartine intervalului\
            [0,170]\n");
    exit(1);
 }
 for( f=1.0, i=2; i <= n; i++)
    f *= i;
 printf("n =%d\tn! =%g\n", n,f);
}
```

4.19 Să se scrie un program care citeşte pe *n* şi *a*, calculează şi afişează valoarea lui *a* la puterea a *n*-a (a**n). *n* este întreg nenegativ, iar a este un număr oarecare. Calculele se fac în flotantă *long double*.

O metodă simplă pentru a ridica pe *a* la puterea a *n*-a, pentru *n* natural, este de a înmulţi pe *a* cu el însuşi de *n* ori.

PROGRAMUL BIV19

```
#include <stdio.h>
#include <stdlib.h>

main() /* citeste pe n si a, calculeaza si afiseaza pe a**n */
{
 int n,i;
 long double a,p;

 printf("Baza a = ");
 if(scanf("%Lf", &a ) != 1) {
    printf("nu s-a tastat un numar\n");
    exit(1);
 }
 printf("Exponentul n = ");
```

```
    if(scanf("%d", &n ) != 1 || n<0 ) {
        printf("nu s-a tastat un intreg nenegativ\n");
        exit(1);
    }
    for(i=0,p=1.0L; i < n ; i++ )
        p *= a;
    printf("a=%Lg\tn=%d\ta**n=%Lg\n",a,n,p );
}
```

Observaţii:

1. Funcţia *pow*, de prototip:

   double pow(double *x*,**double** *y*);

   se utilizează pentru date de tip *double*.

2. Metoda utilizată în programul de faţă nu este eficientă pentru *n* relativ mare, deoarece necesită multe înmulţiri. Numărul lor poate fi redus substanţial procedînd ca mai jos.

Fie

f=a

Executînd repetat instrucţiunea de atribuire

f=f*f;

se generează puterile:

(1) a**2, a**4, a**8, a**16, a**32, ...

adică exponenţii lui *a* sînt puteri ale lui 2.

Termenii şirului (1) se generează printr-un număr relativ mic de operaţii de înmulţire. a**n se poate exprima folosind numai factori din şirul (1) şi eventual şi pe *a*, dacă *n* este impar. Acest lucru rezultă din faptul că orice număr natural pozitiv, se poate exprima ca o sumă de puteri ale lui 2. Într-adevăr, nu avem decît să reprezentăm numărul *n* în binar şi să însumăm puterile lui doi din şirul:

(2) 1,2,4,8,16,32,64,...

care corespund cifrelor 1 din această reprezentare.

Corespondenţa dintre termenii şirului (2) şi cifrele reprezentării binare se face de la dreapta spre stînga adică 1 se pune în corespondenţă cu ultima cifră a reprezentării binare, 2 cu penultima, 4 cu antepenultima etc.

Exemplu:

Fie n=43. În binar *n* se reprezintă astfel:

n=101011

Rezultă că *n* este suma termenilor 1, 2, 8 şi 32 aleşi din şirul (2) conform cifrelor 1 din reprezentarea binară a lui:

n=1+2+8+32=43

Pentru a calcula pe a**n va fi suficient să realizăm produsul factorilor a**2, a**8, şi a**32, factori aleşi din şirul

(1) şi a.

Din cele de mai sus rezultă că pentru a reduce numărul înmulţirilor la calculul lui a**n este suficient să generăm factorii şirului (1) şi să-i alegem pe cei ai căror exponenţi însumaţi ne dau pe *n*. Selectarea lor se poate face simplu pornind de la reprezentarea binară a lui *n*. Astfel, dacă ultima cifră binară a lui *n* este 1, atunci se selectează ca prim factor chiar *a*. Apoi, făcînd o deplasare spre dreapta a lui *n* cu o poziţie binară, se alege sau nu factorul a**2, după cum ultima cifră binară a lui *n*, după ce s-a făcut deplasarea, este 1 sau nu. Procedeul continuă pînă cînd *n* devine egal cu zero.

Procesul de calcul se defineşte astfel:

1. p=1
2. f=a
3. Cît timp *n* este diferit de zero se execută:

3.1. Dacă ultima cifră binară a lui *n* este egală cu 1, atunci p=p*f

3.2. Se generează termenul următor din şirul (1):

f=f*f

3.3. *n* se deplasează spre dreapta cu o poziţie binară.

Se observă că punctul 3.2. este necesar numai dacă n>1. Într-adevăr, dacă n=1 atunci procesul de calcul se termină, deoarece prin deplasarea lui *n* la dreapta cu o poziţie binară acesta devine egal cu zero şi deci ciclul nu se mai reia. Avînd în vedere faptul că termenii şirului (1) cresc rapid, este important ca să suprimăm termenul care s-ar genera pentru n=1, deoarece cu toate că el nu este necesar, calculul lui poate conduce la o depăşire flotantă superioară. În acest fel, punctul 3.2. se schimbă cu

dacă n>1 atunci

f=f*f

Programul BIV19A ridică pe *a* la *n* folosind procesul de calcul descris mai sus.

PROGRAMUL BIV19A

```
#include <stdio.h>
#include <stdlib.h>

main() /* citeste pe n si a, calculeaza si afiseaza pe a**n */
{
 int n,i;
 long double a,p,f;

 printf("Baza a = ");
 if(scanf("%Lf", &a ) != 1) {
     printf("nu s-a tastat un numar\n");
     exit(1);
 }
 printf("Exponentul n = ");
 if(scanf("%d", &n ) != 1 || n<0 ) {
     printf("nu s-a tastat un intreg nenegativ\n");
     exit(1);
 }
 i = n;
 for(p=1.0L,f = a;  n ; n >>= 1 ) {
     if( n & 1 )
         p *= f; /* factor selectat din sirul (1) */
     if( n > 1 )
         f *= f; /* genereaza termenul urmator din sirul (1) */
 }
 printf("a=%Lg\tn=%d\ta**n=%Lg\n",a,i,p );
}
```

Observaţii:

1. *exp1* realizează iniţializările de la punctele 1 şi 2 din descrierea de mai sus:

   p=1.0L, f=a

2. *exp2*, condiţia de continuare a ciclului s-a redus la *n*; deci ciclul continuă atît timp cît *n* este *adevărat*, adică este diferit de zero.

3. Punctul 3.3 este partea de reiniţializare a ciclului *for* şi ea constă din expresia

n >>= 1

care deplasează pe *n* cu un ordin binar spre dreapta.

4. Expresia

   n&1

   are valoarea 1, dacă ultimul ordin binar al lui *n* este egal cu 1. În acest caz factorul generat în variabila *f* se selectează, deci se înmulţeşte cu *p*.

4.8. Instrucţiunea do-while

Instrucţiunea *do-while* realizează structura *ciclică condiţionată posterior*. Această instrucţiune poate fi realizată cu ajutorul celorlalte instrucţiuni definite pînă în prezent. Cu toate acestea, prezenţa ei în limbaj măreşte flexibilitatea în programare.

Ea are formatul:

do
instrucţiune
while(*expresie*);

Instrucţiunea *do-while* se execută în felul următor:

1. Se execută *instrucţiune*
2. Se evaluează *expresie*; dacă aceasta are o valoare diferită de zero, atunci se revine la punctul 1, pentru a executa din nou *instrucţiune*; altfel (expresia are valoarea zero) se trece în secvenţă, adică la instrucţiunea următoare instrucţiunii *do-while*.

Se observă că în cazul acestei instrucţiuni întîi se execută *instrucţiune* şi apoi se testează condiţia de repetare a execuţiei ei.

Instrucţiunea *do-while* este echivalentă cu secvenţa:

instrucţiune
while(*expresie*)
instrucţiune

Considerăm şi în acest caz că, *instrucţiune* reprezintă corpul ciclului *do-while*.

În cazul instrucţiunii *do-while* corpul ciclului se execută cel puţin o dată, spre deosebire de cazul instrucţiunilor *while* şi *for*, cînd este posibil să nu se execute niciodată.

Exerciţii:

4.20 Să se scrie un program care citeşte un număr nenegativ, subunitar, calculează şi afişează rădăcina pătrată din numărul respectiv cu o eroare mai mică decît 0,0000000001.

Programul de faţă nu foloseşte funcţia *sqrt* din biblioteca standard.

Pentru a extrage rădăcina pătrată dintr-un număr *a* din intervalul (0,1), se poate folosi metoda iterativă a lui Newton de mai jos.

Fie şirul:

x[0], x[1], ..., x[n], ...

unde:

(1) x[n+1]=0,5(x[n]+a/x[n]) pentru n=0,1,2,...

Se demonstrează că şirul de mai sus este convergent şi are limita egală cu rădăcina pătrată din *a*. Convergenţa este foarte rapidă pentru *a* subunitar. În acest caz se poate lua x[0]=1.

Pentru a obţine rădăcina pătrată cu precizia cerută, este suficient ca diferenţa, în valoare absolută, dintre doi termeni consecutivi ai şirului să fie mai mică decît

EPS=0.0000000001

Să observăm că pentru a rezolva problema cu metoda indicată mai sus, la fiecare pas al calculului, sînt necesari numai doi termeni ai şirului:

a. Din x[n] se determină x[n+1] cu relaţia (1).

b. Se compară

abs(x[n]-x[n+1]) cu EPS;

Dacă valoarea absolută respectivă nu este mai mică decît EPS, atunci se calculează x[n+2] folosind relaţia (1), în care x[n] se înlocuieşte cu x[n+1].

Deci se poate reveni la punctul *a* de mai sus, după ce lui x[n] i se atribuie valoarea x[n+1].

Deci, în loc să utilizăm tabloul *x*, este suficient să folosim două variabile *x1* şi *x2* care păstrează în fiecare moment doi termeni consecutivi ai şirului de mai sus. Termenii şirului se obţin executînd repetat secvenţa de atribuiri:

```
x1=x2;
x2=0.5*(x1+a/x1);
```

După fiecare execuţie a acestei secvenţe, *x1* şi *x2* au ca valori doi termeni consecutivi ai şirului căutat.

Secvenţa se va repeta atît timp cît (2) abs(x1-x2) >= EPS, test care îl facem după fiecare execuţie a secvenţei de atribuiri. Dacă el se confirmă, atunci se repetă secvenţa respectivă, altfel valoarea variabilei *x2* aproximează rădăcina pătrată din *a* cu o eroare mai mică decît EPS.

Acest proces de calcul se realizează imediat printr-o instrucţiune *do-while*.

Secvenţa de atribuiri este corpul ciclului, iar expresia (2) reprezintă condiţia de continuare a lui. Întrucît primul termen al şirului are valoarea 1, instrucţiunea *do-while* este precedată de atribuirea:

x2=1;

PROGRAMUL BIV20

```
#include <stdio.h>
#include <stdlib.h>

#define EPS 1e-10

main() /* citeste pe a din intervalul [0,1], calculeaza si
          afiseaza radacina patrata din a */
{
 double a,x1,x2,y;

 printf("Tastati valoarea lui a = ");
 if(scanf("%lf",&a) != 1 || a < 0 || a > 1) {
    printf("Nu s-a tastat un numar in\
          intervalul [0,1]\n");
    exit(1);
 }
 if( a == 0 )
    x2 = 0; /* radacina patrata din zero este zero */
 else
    if( a == 1)
       x2 = 1; /* radacina patrata din 1 este 1 */
    else { /* a este diferit de 0 sau 1*/
       x2 = 1;
       do {
          x1 = x2;
          x2 = 0.5*(x1 + a/x1);
```

```
            if(( y = x1 - x2) < 0 )
                    y = -y;
/* y = abs(x1-x2)*/

        } while( y >= EPS );
    } /* sfirsit else */
 printf("a=%.11g\tsqrt(a)=%.11g\n",a,x2);
}
```

Observaţii:

1. Instrucţiunea *do-while* poate fi scrisă mai compact astfel:

```
do {
    x1=x2;
    x2=0.5*(x1+a/x1);
}while((y=x1-x2)<0?-y:y,y>=EPS);
```

2. Pentru numere supraunitare se poate aplica aceeaşi metodă. În acest caz se poate lua ca prim termen al şirului chiar numărul din care se extrage rădăcina pătrată. O altă posibilitate este de a extrage rădăcina pătrată din inversul numărului. Aceasta deoarece metoda este rapid convergentă pentru numere subunitare.

   Fie a>1. Atunci x=1/a. Dacă y este rădăcina pătrată din *x*, atunci 1/y este rădăcina pătrată din *a*.

4.21 Să se scrie un program care citeşte un întreg pozitiv ce reprezintă o sumă exprimată în lei. Se cere să se afişeze numărul minim de bancnote şi monede de 1000 lei, 500 lei, 200 lei etc. necesare pentru a exprima suma respectivă.

Procesul de calcul începe cu determinarea numărului de bancnote de 1000 de lei. În acest scop se împarte suma respectivă *s* la 1000:

s/1000

Pentru a determina numărul bancnotelor de 500 de lei se determină restul împărţirii lui *s* la 1000 şi acest rest se împarte la 500 şi aşa mai departe. Se obţin paşii următori:

1. q=s/1000; *afişează q*
2. s=s%1000;
3. q=s/500; *afişează q*

4. s=s%500;

5. q=s/200; *afişează q*

6. s=s%200;

7. q=s/100; *afişează q*

8. s=s%100;

9. q=s/50; *afişează q*

10. s=s%50;

11. q=s/20; *afişează q*

12. s=s%20;

13. q=s/10; *afişează q*

14. s=s%10;

15. q=s/5; *afişează q*

16. s=s%5;

17. q=s/1; *afişează q*

18. s=s%1.

Se observă că secvenţa:

q=s/n; *afişează q*
s=s%n;

se repetă pentru valorile lui *n* egale cu

1000, 500, 200, 100, 50, 20, 10, 5 şi 1

Pentru a putea executa repetat această secvenţă este nevoie de o metodă simplă prin care să se atribuie lui *n* valorile corespunzătoare. Aceasta se poate realiza dacă păstrăm datele respective într-un tablou de tip *int*. Fie:

mon[0]=1, mon[1]=5, mon[2]=10,
mon[3]=20, mon[4]=50, mon[5]=100,
mon[6]=200, mon[7]=500, mon[8]=1000.

Atunci *n* poate fi înlocuit prin mon[i], unde iniţial i=8 şi se decrementează la fiecare pas al ciclului. Deci urmează a se executa repetat secvenţa:

q=s/mon[i]; *afişează q*
s=s%mon[i];
i=i-1;

Această secvenţă se va executa repetat pînă cînd *s* devine zero.

PROGRAMUL BIV21

```
#include <stdio.h>
#include <stdlib.h>

main() /* exprima o suma de lei in bancnote si monede de
         1000 lei, 500 lei etc. */
{
 int mon[9];
 long s,q;
 int i;

 printf("Tastati suma: ");
 if(scanf("%ld",&s) != 1 || s <= 0 ) {
    printf("nu s-a tastat un intreg pozitiv\n");
    exit(1);
 }
 mon[0] = 1; mon[1] = 5; mon[2] =10; mon[3] = 20;
 mon[4] = 50; mon[5] = 100; mon[6] = 200;
 mon[7] =500; mon[8] = 1000;
 i = 8;
 do {
    q = s/mon[i];
    if(q ) /* afisarea se face daca q este diferit de zero */
        if( i < 6 ) /* monede */
            if( q == 1 ) /* exista o singura moneda */
                if( i == 0 ) /* o moneda de 1 leu */
                    printf("o moneda de 1 leu\n");
            else
                printf("o moneda a %d lei\n",
                        mon[i]);
        else /* mai multe monede */
            if( i== 0 ) /* monede de 1 leu */
                printf("%ld monede de 1 leu\n",q);
            else /* monede a mon[i] lei */
                printf("%ld monede a %d lei\n",
                        q, mon[i]);
      else /* bancnote */
        if( q == 1 ) /* o singura bancnota */
            printf("o bancnota a %d lei\n",mon[i]);
        else /* mai multe bancnote */
            printf("%ld bancnote a %d lei\n",q,
                    mon[i] );
```

```
    } while ( s %= mon[i--]);
}
```

4.9. Instrucţiunea continue

Această instrucţiune se poate utiliza numai în corpul unui ciclu. Ea permite abandonarea iteraţiei curente. Formatul ei este:

continue;

Efectul instrucţiunii *continue* este următorul:

a. În corpul instrucţiunilor *while* şi *do-while*:

   La întîlnirea instrucţiunii *continue* se abandonează iteraţia curentă şi se trece la evaluarea *expresiei* care stabileşte continuarea sau terminarea ciclului respectiv (expresia inclusă între paranteze rotunde şi care urmează după cuvîntul cheie *while*).

b. În corpul instrucţiunii *for*:

   La întîlnirea instrucţiunii *continue* se abandonează iteraţia curentă şi se trece la execuţia pasului de reiniţializare.

Instrucţiunea *continue* nu este o instrucţiune obligatorie. Prezenţa ei măreşte flexibilitatea în scrierea programelor C. Ea conduce adesea la diminuarea nivelelor de imbricare ale instrucţiunilor *if* utilizate în corpul ciclurilor.

4.10. Funcţiile standard sscanf şi sprintf

Biblioteca standard a limbajelor C şi C++ conţine funcţiile *sscanf* şi *sprintf* care sînt analoge funcţiilor *scanf* şi respectiv *printf*. Ele au un parametru în plus la apel şi anume primul lor parametru este adresa unei zone de memorie în care se pot păstra caractere ale codului ASCII. Ceilalţi parametrii sînt indentici cu cei întîlniţi în corespondentele lor, *scanf* şi respectiv *printf*.

Primul parametru al acestor funcţii poate fi numele unui tablou de tip *char*, deoarece un astfel de nume are ca valoare chiar adresa de început a zonei de memorie care îi este alocată.

Funcţia *sprintf* se foloseşte, ca şi funcţia *printf*, pentru a realiza conversii ale datelor de diferite tipuri din formatele lor interne, în formate externe reprezentate prin succesiuni de caractere. Diferenţa constă în aceea că, de data aceasta caracterele respective nu se afişează pe terminalul standard, ci se păstrează în zona de memorie definită de primul parametru al funcţiei

sprintf. Ele se păstrează sub forma unui şir de caractere şi pot fi afişate ulterior din zona respectivă cu ajutorul funcţiei *puts*. De aceea, un apel al funcţiei *printf* poate fi totdeauna înlocuit cu un apel al funcţiei *sprintf*, urmat de un apel al funcţiei *puts*. O astfel de înlocuire este utilă cînd dorim să afişăm de mai multe ori aceleaşi date. În acest caz se apelează funcţia *sprintf* o singură dată pentru a face conversiile necesare din format intern în format extern, rezultatele conversiilor păstrîndu-se într-un tablou de tip caracter. În continuare se pot afişa datele respective apelînd funcţia *puts* ori de cîte ori este necesară afişarea lor. Funcţia *sprintf*, ca şi funcţia *printf* returnează numărul octeţilor şirului de caractere rezultat în urma conversiilor efectuate.

Exemplu:

```
int zi,luna,an;
char data_calend[11];
...
sprintf(data_calend,"%02d/%02d/%d",zi,luna,an);
puts(data_calend);
...
puts(data_calend);
```

Funcţia *sscanf* realizează, ca şi funcţia *scanf*, conversii din format extern în format intern. Deosebirea constă în faptul că de data aceasta caracterele nu sînt citite din zona tampon corespunzătoare tastaturii, ci ele provin dintr-o zonă de memorie a cărei adresă este definită de primul parametru al funcţiei *sscanf*. Aceste caractere pot proveni în zona respectivă în urma apelului funcţiei *gets*. În felul acesta, apelul funcţiei *scanf* poate fi înlocuit prin apelul funcţiei *gets* urmat de apelul funcţiei *sscanf*. Astfel de înlocuiri sînt utile cînd dorim să eliminăm eventualele erori apărute la tastarea datelor.

Funcţia *sscanf*, ca şi funcţia *scanf*,returnează numărul cîmpurilor convertite corect conform specificatorilor de format prezenţi în parametrul de control. La întîlnirea unei erori, ambele funcţii îşi întrerup execuţia şi se revine din ele cu numărul cîmpurilor tratate corect. Analizînd valoarea returnată, se poate stabili dacă au fost prelucrate corect toate cîmpurile sau a survenit o eroare.

În caz de eroare se poate reveni pentru a introduce corect datele respective. În acest scop este necesar să se elimine caracterele începînd cu cel din poziţia eronată.

În cazul în care se utilizează secvenţa:

```
gets
sscanf
```

abandonarea caracterelor respective se face automat reapelîndu-se funcţia *gets*. În cazul utilizării funcţiei *scanf* este necesar să se avanseze pînă la caracterul *newline* aflat în zona tampon ataşată tastaturii sau să se videze zona respectivă prin funcţii speciale.

În exerciţiile care urmează vom folosi secvenţele formate din apelurile funcţiei *gets* urmate de apelurile lui *sscanf*. O astfel de secvenţă se apelează repetat în cazul în care se întîlnesc erori în datele de intrare.

Exemplu:

```
char tab[255];
int zi,luna,an;
...
gets(tab);
sscanf(tab,"%d %d %d",&zi,&luna,&an);
```

Amintim că funcţia *gets* returnează valoarea *NULL* la întîlnirea sfîrşitului de fişier.

Funcţiile *sscanf* şi *sprintf* au prototipurile în fişierul *stdio.h*.

Exerciţii:

4.22 Să se scrie un program care citeşte un întreg pozitiv de tip *long*, stabileşte dacă acesta este prim şi afişează un mesaj corespunzător.

PROGRAMUL BIV22

```
#include <stdio.h>
#include <stdlib.h>

main() /* citeste un intreg pozitiv de tip long, stabileste daca
          acesta este un numar prim si afiseaza un mesaj
          corespunzator */
{
 long n;
 long i;
 int j;
 char tab[255];

 do {
     printf("tastati un intreg pozitiv:");
     if(gets(tab) == NULL ) {
       printf("s-a tastat EOF\n");
       exit(1);
     }
```

```
        if(sscanf(tab,"%ld",&n) != 1 || n <= 0 ) {
           printf("nu s-a tastat un intreg\
                  pozitiv\n");
           j = 1;
           continue; /* se va relua ciclul deoarece se trece la
                        evaluarea expresiei j care este diferita
                        de zero */
        }
        j = 0; /* ciclul se intrerupe deoarece s-a citit corect
                  un intreg pozitiv */
     } while (j);
     for( j = 1, i = 2; i*i <= n && j; i++)
        if(n%i == 0) /* numarul nu este prim */
           j = 0;
     printf("numarul: %ld", n);
     if( j == 0 )
        printf(" nu ");
     printf(" este prim\n");
   }
```

Observaţii:

1. Utilizarea instrucţiunii *continue* se poate omite folosind o instrucţiune *if* cu alternativă *else*:

```
if(sscanf(...) !=1||n<=0){
   ...
   j=1;
}else
   j=0;
```

2. Ciclul *for* continuă atît timp cît expresia:

   i*i<=n&&j

   este adevărată. Această expresie se evaluează de la stînga spre dreapta şi din această cauză expresia

   i*i<=n

   se evaluează şi atunci cînd j=0. De aceea expresia respectivă este mai eficientă sub forma:

   j&&i*i<=n

   În acest caz, pentru j=0 nu se mai evaluează restul expresiei.

Expresia:

i*i<=n

rezultă din faptul că, pentru a stabili că un număr este prim, este suficient să încercăm divizibilitatea lui prin numere al căror pătrat nu-l depăşeşte pe *n*.

O altă posibilitate este aceea de a înlocui expresia

i*i<=n

cu

i<=sqrt(n)

În acest caz este util să se calculeze rădăcina pătrată în afara ciclului *for*:

```
...
long q;
...
q=sqrt((double)n);
for(j=1,i=2;j&&i<=q;i++)
 if(n%i==0)
     j=0;
...
```

O altă simplificare posibilă este schimbarea instrucţiunii *if* din corpul ciclului *for* cu:

j=n%i;

4.11. Instrucţiunea break

Instrucţiunea *break* este înrudită cu instrucţiunea *continue*. Ea are formatul:

break;

Poate fi utilizată în corpul unui ciclu. În acest caz, la întîlnirea instrucţiunii *break* se termină execuţia ciclului în al cărui corp este inclusă şi execuţia continuă cu instrucţiunea următoare instrucţiunii ciclice respective.

Instrucţiunea *break*, la fel ca şi instrucţiunea *continue*, măreşte flexibilitatea la scrierea programelor în limbajele C şi C++.

Exerciţii:

4.23 Să se scrie un program care citeşte măsurile *a, b, c* ale laturilor unui triunghi, calculează şi afişează aria triunghiului respectiv.

Pentru determinarea ariei se va folosi formula lui Heron. Dacă notăm cu *p* semiperimetrul triunghiului, atunci aria se calculează cu ajutorul expresiei:

aria=sqrt(p*(p-a)*(p-b)*(p-c))

Amintim că măsurile laturilor triunghiului satisfac relaţiile

p-a > 0, p-b > 0, şi p-c > 0

deci se va testa condiţia

p-a > 0 && p-b > 0 && p-c > 0

care trebuie să fie adevărată.

PROGRAMUL BIV23

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

main() /* citeste masurile laturilor unui triunghi si afiseaza aria lui */
{
 double a,b,c,p;
 char tab[255];

 do { /* citeste masurile laturilor triunghiului */
     do { /* citeste pe a */
         printf("a= ");
         if(gets(tab) == NULL ) {
            printf("s-a tastat EOF\n");
            exit(1);
         }
         if(sscanf(tab,"%lf",&a) == 1 && a>0)
            break; /* se iese din ciclul pentru citirea lui a */
         printf("nu s-a tastat un numar pozitiv\n");
         printf("se reia citirea lui a\n");
     } while (1);
     do { /* citeste pe b */
         printf("b= ");
         if(gets(tab) == NULL ) {
            printf("s-a tastat EOF\n");
            exit(1);
         }
         if(sscanf(tab,"%lf",&b) == 1 && b>0)
```

```
            break;  /* se iese din ciclul do-while deoarece s-a
                       citit valoarea lui b */
        printf("nu s-a tastat un numar pozitiv\n");
        printf("se reia citirea lui b\n");
    } while (1);
    do {
        printf("c= ");
        if(gets(tab) == NULL ) {
            printf("s-a tastat EOF\n");
            exit(1);
        }
        if(sscanf(tab,"%lf",&c ) == 1 && c>0)
            break;  /* se iese din ciclul do-while deoarece
                       s-a citit valoarea lui c */
        printf("nu s-a tastat un numar pozitiv\n");
        printf("se reia citirea lui c\n");
    } while(1);
    p = (a + b + c) / 2;
    if( p-a > 0 && p -b > 0 && p - c >0 )
        break;  /* se iese din ciclul do-while exterior deoarece a,b,c
                   pot fi masurile laturilor unui triunghi */
    printf("a= %g\tb =%g\tc= %g\n",a,b,c);
    printf("nu pot reprezenta masurile laturilor\
          unui triunghi\n");
  } while (1);
 printf("aria = %g\n",sqrt(p*(p-a)*(p-b)*(p-c)) );
}
```

4.24 Să se scrie un program care citeşte două numere naturale şi pozitive, calculează şi afişează cel mai mare divizor comun al lor (c.m.m.d.c.).

Calculul celui mai mare divizor comun a două numere se poate face folosind algoritmul lui Euclid. Paşii acestui algoritm sînt:

1. Se citesc cele două numere.

   Fie acestea *a* şi *b*.

2. Se împarte *a* la *b*; fie *q* şi *r* cîtul, respectiv restul acestei împărţiri:

   a=q*b+r.

3. Dacă r este diferit de zero, se fac atribuirile:

   a=b şi b=r

   şi se revine la pasul 2.

Altfel algoritmul se întrerupe şi cel mai mare divizor comun al lor este *b*.

PROGRAMUL BIV24

```
#include <stdio.h>
#include <stdlib.h>

main() /* citeste doua numere intregi si pozitive
          si afiseaza c.m.m.d.c al lor */
{
 long m,n;
 long a,b;
 long r;
 char tab[255];

 do {
    printf("primul numar= ");
    if(gets(tab) == NULL ) {
        printf("s-a tastat EOF\n");
        exit(1);
    }
    if(sscanf(tab,"%ld",&m) == 1 && m > 0 )
        break;
    printf("nu s-a tastat un intreg pozitiv\n");
 } while(1);

 do {
    printf("al doilea numar= ");
    if(gets(tab) == NULL ) {
        printf("s-a tastat EOF\n");
        exit(1);
    }
    if(sscanf(tab,"%ld",&n) == 1 && n > 0 )
        break;
    printf("nu s-a tastat un intreg pozitiv\n");
 } while(1);

 a = m;
 b = n;

 do {
    r = a % b;
```

```
        if(r) {
            a = b;
            b = r;
        }
    } while(r);

    printf("(%ld,%ld) = %ld\n", m,n,b);
}
```

4.12. Instrucţiunea switch

Instrucţiunea *switch* permite realizarea structurii *selective*. Aceasta este o generalizare a structurii *alternative* şi a fost introdusă de C.A.R. Hoare. Ea poate fi realizată prin instrucţiuni *if* imbricate. Utilizarea instrucţiunii *switch* face ca programul să fie mai clar decît dacă se utilizează varianta cu instrucţiuni *if* imbricate.

Structura *selectivă*, în forma în care a fost ea acceptată de către adepţii programării structurate, se realizează în limbajul C cu ajutorul următorului format al instrucţiunii *switch*:

switch(*expresie*){
  case c1:
      sir_1
      break;
  case c2 :
      sir_2
      break;
...
  case cn :
      sir_n
      break;
  default:
      sir
}

unde:

c1, c2, ..., cn	- Sînt constante.
sir_1, sir_2, ..., sir_n	- Sînt succesiuni de instrucţiuni.

Instrucţiunea *switch* cu formatul indicat mai sus se execută astfel:

1. Se evaluează expresia din parantezele rotunde.

2. Se compară pe rînd valoarea expresiei cu valorile constantelor c1, c2, ..., cn.

   - Dacă valoarea expresiei coincide cu una din constante să zicem cu *ci*, atunci se execută secvenţa *sir_i*, apoi se trece la instrucţiunea următoare instrucţiunii *switch*, adică la instrucţiunea aflată după acolada închisă care termină instrucţiunea.

   - Dacă valoarea respectivă nu coincide cu nici una din constantele c1, c2, ..., cn, atunci se execută succesiunea de instrucţiuni *sir* şi apoi se ajunge la instrucţiunea următoare instrucţiunii *switch*.

Menţionăm că este posibil să nu se ajungă la instrucţiunea următoare instrucţiunii *switch* în cazul în care succesiunea de instrucţiuni selectată pentru execuţie (*sir_i* sau *sir*) va defini ea însăşi un alt mod de continuare a execuţiei programului (de exemplu, execuţia instrucţiunii de revenire dintr-o funcţie, saltul la o instrucţiune etichetată etc.)

Succesiunile *sir, sir_1, ..., sir_n* se numesc *alternativele* instrucţiunii *switch*.

Alternativa *sir* este opţională, deci într-o instrucţiune *switch*, secvenţa

default:
sir

poate fi absentă.

În acest caz, dacă valoarea expresiei nu coincide cu valoarea nici uneia dintre constantele *c1,c2,...,cn*,atunci instrucţiunea *switch* nu are nici un efect şi se trece la execuţia instrucţiunii următoare.

Instrucţiunea *switch* de mai sus este echivalentă cu următoarea instrucţiune *if* imbricată:

```
if(expresie==c1)
 sir_1
else
 if( expresie==c2   )
     sir_2
 else
     if( expresie==c3   )
          sir_3
     else
          if
               ...
```

```
else
          if( expresie==cn  )
                    sir_n
          else
                    şir
```

Instrucţiunea *break* de la sfîrşitul fiecărei alternative, după secvenţele *sir_1, sir_2, ..., sir_n*, permite ca la întîlnirea ei să se treacă la execuţia instrucţiunii următoare instrucţiunii *switch*. Se obişnuieşte să se spună că instrucţiunea *break* permite ieşirea din instrucţiunea *switch*.

Amintim că instrucţiunea *break* poate fi utilizată numai în corpurile ciclurilor şi în alternativele instrucţiunii *switch*.

Prezenţa ei la sfîrşitul fiecărei alternative a unei instrucţiuni *switch*, nu este obligatorie. În cazul în care instrucţiunea *break* este absentă la sfîrşitul unei alternative, după execuţia succesiunii de instrucţiuni din compunerea alternativei respective se va trece la execuţia succesiunii de instrucţiuni din alternativa următoare a aceleeaşi instrucţiuni *switch*.

Astfel, dacă o instrucţiune *switch* are formatul:

```
switch(expresie){
 case c1 :
      sir_1
 case c2 :
      sir_2
}
```

atunci ea este echivalentă cu următoarea secvenţă:

```
if(expresie==c1){
 sir_1
 sir_2
}else
 if( expresie==c2  ){
      sir_2
 }
```

Exerciţii:

4.25 Să se scrie un program care citeşte o cifră din intervalul [1,7] şi afişează denumirea zilei din săptămînă corespunzătoare cifrei respective (1-luni, 2-marţi etc).

PROGRAMUL BIV25

```
#include <stdio.h>
#include <stdlib.h>

main() /* citeste o cifra din intervalul [1,7] si afiseaza denumirea
          zilei din saptamina corespunzatoare cifrei respective */
{
 int i;
 char t[255];

 do {
     puts("tastati o cifra din intervalul [1,7]");
     if(gets(t) == NULL ){
         puts("s-a tastat EOF");
         exit(1);
     }
     if(sscanf(t,"%d", &i) == 1 && i > 0 && i < 8 )
         break;
     puts("nu s-a tastat o cifra din\
             intervalul  [1,7]");
 } while(1);
 switch(i) {
     case 1:
         puts("luni");
         break;
     case 2:
         puts("marti");
         break;
     case 3:
         puts("miercuri");
         break;
     case 4:
         puts("joi");
         break;
     case 5:
         puts("vineri");
         break;
     case 6:
         puts("simbata");
         break;
     case 7:
         puts("duminica");
 }
```

}

4.26 Să se scrie un program care citeşte construcţii de forma:

op1 op op2

unde:

op1 şi op2 - Sînt întregi de tip *long*.

op - Este unul din caracterele: +, -, * sau /.

şi afişează valoarea expresiei citite.

Operaţia de împărţire este împărţirea întreagă.

Acest exemplu este întîlnit frecvent în diferite lucrări relativ la limbajul C. De exemplu, autorii limbajului C, în lucrarea [2] propun acest exemplu pentru a simula un calculator rudimentar de birou.

PROGRAMUL BIV26

```
#include <stdio.h>
#include <stdlib.h>

main() /* citeste o expresie de forma: op1opop2
          unde op1 si op2 sînt intregi de tip long, iar op este unul din
          operatorii  +, -, *, /
          afiseaza valoarea expresiei respective */
{
 long op1,op2,rez;
 char op;
 char t[255];

 for ( ; ; ) {
     do {
         printf("tastati expresia:op1opop2: ");
         if( gets(t) == NULL )
             exit(0);
       . if(sscanf(t,"%ld %c %ld",&op1,&op,&op2)
              == 3)
             break;
         printf("expresie eronata\n");
     } while(1);

     switch(op) {
         case '+' :
```

```
            rez=op1 + op2;
            break;
        case '-' :
            rez=op1 - op2;
            break;
        case '*' :
            rez= op1 * op2;
            break;
        case '/' :
            if(op2 == 0 ) {
                printf("divizor nul\n");
                rez = 0;
            }
            else
                rez = op1/op2;
            break;
        default:
            printf("operator eronat\n");
            rez = 0;
    } /* sfirsit switch*/

    printf("op1= %ld op= %c op2= %ld rez=%ld\n",
            op1,op,op2,rez);
  } /* sfirsit for */
} /* sfirsit main */
```

Observaţie:

Programul de faţă permite să se evalueze mai multe expresii separate prin caractere albe. El îşi întrerupe execuţia la întîlnirea sfîrşitului de fişier.

4.13. Instrucţiunea goto

Instrucţiunea *goto* nu este o instrucţiune absolut necesară la scrierea programelor în limbajul C. Cu toate acestea, ea se dovedeşte utilă în anumite cazuri. Autorii limbajului recomandă utilizarea ei în cazul în care se doreşte să se iasă din mai multe cicluri imbricate. Astfel de situaţii apar adesea la întîlnirea unei erori. În astfel de situaţii, de obicei, se doreşte să se facă un salt în afara ciclurilor în care a intervenit eroarea, pentru a se ajunge la o secvenţă externă lor de tratare a erorii respective.

Înainte de a indica formatul instrucţiunii *goto* să precizăm noţiunea de *etichetă*.

Prin *etichetă* înţelegem un nume urmat de două puncte:

nume:

nume utilizat în definirea unei etichete este numele etichetei respective.

După o etichetă urmează o instrucţiune.

Se obişnuieşte să se spună că eticheta *prefixează* instrucţiunea care urmează după ea.

Etichetele sînt *locale* în corpul funcţiei în care sînt definite.

Instrucţiunea *goto* are formatul:

goto *nume*;

unde:

nume - Este numele unei etichete definită în corpul aceleeaşi funcţii în care se află instrucţiunea *goto*.

La întîlnirea instrucţiunii *goto*, se realizează un salt la instrucţiunea prefixată de eticheta al cărei nume se află după cuvîntul cheie *goto*.

Deoarece o etichetă este locală în corpul unei funcţii rezultă că ea este nedefinită în afara corpului funcţiei respective. În felul acesta, o instrucţiune *goto* poate realiza un salt numai la o instrucţiune din corpul aceleeaşi funcţii în care este ea utilizată.

Deci, o instrucţiune *goto* nu poate face salt din corpul unei funcţii la o instrucţiune din corpul altei funcţii.

Menţionăm că nu se justifică utilizarea abuzivă a acestei instrucţiuni. Se recomandă a fi utilizată pentru a simplifica ieşirea din cicluri imbricate.

Exemplu:

Presupunem că într-un punct al programului, aflat în interiorul mai multor cicluri, se depistează o eroare şi se doreşte să se continue execuţia programului cu o secvenţă de tratare a erorii respective. În acest caz, vom folosi o instrucţiune *goto* ca mai jos.

```
for(...){
 ...
 while(...){
     ...
     do{
         ...
         for(...){
             ...
```

```
            if(i==0)
                goto divzero;
            else
                x=y/i;
                ...
        }
        ...
    }while(...);
    ...
 }
 ...
}
...

/* secventa de tratare a erorii */
divzero:
 printf(...);
...
```

În absenţa instrucţiunii *goto* se poate realiza acelaşi lucru folosind un indicator şi o serie de teste realizate asupra lui.

4.14. Programarea procedurală, funcţii, apelul şi revenirea din ele

Începînd cu primele limbaje de programare de nivel înalt s-a utilizat *programarea procedurală*.

Aceasta s-a dezvoltat din necesitatea de a utiliza într-un program, aceeaşi secvenţă de calcul de mai multe ori. Pentru a evita o astfel de repetiţie, secvenţa de instrucţiuni corespunzătoare se organizează ca o parte distinctă şi se face un salt la ea, ori de cîte ori este nevoie în program de procesul de calcul respectiv. Acest salt este cu revenire la instrucţiunea următoare instrucţiunii care a făcut saltul şi de aceea el diferă de salturile realizate prin instrucţiunea *goto*. Secvenţa de instrucţiuni organizată în acest fel are diferite denumiri în diverse limbaje de programare: subprogram, subrutină, procedură etc. Pentru început considerăm denumirea de *procedură*.

Uneori procedura trebuie să exprime acelaşi proces de calcul dar care se realizează cu date diferite. În acest caz, procedura trebuie realizată generală, făcînd *abstracţie* de datele respective. De exemplu, pentru a evalua expresia:

(1) 4**10-3**20

putem construi o procedură pentru ridicarea la putere, care să fie generală şi să facă abstracţie de valorile efective pentru bază şi exponent: 4 şi 10 pentru prima ridicare la putere, 3 şi 20 pentru cea de a doua. Această generalizare se realizează considerînd ca fiind variabile atît baza cît şi exponentul, iar valorile lor se precizează la fiecare apel al procedurii implementate în acest fel.

Aceste variabile utilizate pentru a putea implementa o procedură generală şi care se concretizează la fiecare apel al procedurii, se numesc *parametri formali.*

În felul acesta, procedura apare ca un rezultat al unui proces de generalizare necesar implementării ei. Ca orice proces de generalizare ea presupune o abstractizare care se realizează prin utilizarea parametrilor formali.

Valorile de la apel ale parametrilor formali se numesc *parametrii efectivi.*

Programarea *procedurală* are la bază utilizarea procedurilor, iar acestea la rîndul lor realizează o abstractizare prin parametri.

În lucrarea [14] se subliniază şi un alt aspect al abstractizării realizate prin utilizarea procedurilor şi anume acela că o procedură se poate asemăna cu o *cutie neagră*, la care i se transferă date la apel, iar aceasta furnizează rezultatele la revenirea din ea. În momentul apelului, se face abstracţie de metoda de prelucrare a datelor de intrare în rezultate care se mai numesc şi date de ieşire.

În toate limbajele de programare se consideră două categorii de proceduri:

a. Proceduri care definesc o valoare de revenire.

b. Proceduri care nu definesc o valoare de revenire.

Procedurile din categoria a. de obicei se numesc *funcţii. Valoarea de revenire* se mai numeşte şi *valoare de întoarcere* sau *valoarea returnată* de funcţie.

Procedura pentru calculul ridicării la putere este un exemplu de funcţie. Ea are ca parametrii baza şi exponentul, iar ca valoare de întoarcere sau returnată, rezultatul ridicării valorii bazei la valoarea exponentului, valori care sînt definite la apel.

În limbajele C şi C++ atît procedurile din categoria a., cît şi cele din categoria b. se numesc funcţii. Deci, în aceste limbaje distingem funcţii care returnează o valoare la revenirea din ele, precum şi funcţii care nu returnează nici o valoare.

O funcţie are o *definiţie* şi atîtea *apeluri* într-un program, cîte sînt necesare.

O definiţie de funcţie are formatul:

antet
corp

unde:

antet - Are formatul:

tip nume (*lista declaraţiilor parametrilor formali*)

corp - Este o instrucţiune compusă.

Amintim că *tip* este cuvîntul cheie *void* pentru funcţii care nu returnează nici o valoare la revenirea din ele. Pentru alte detalii vezi paragraful 1.4.

Apelul unei funcţii trebuie să fie precedat de definiţia sau de prototipul ei.

Prototipul unei funcţii conţine informaţii asemănătoare cu cele din antetul ei:

- tipul valorii returnate;
- numele funcţiei;
- tipurile parametrilor.

El poate avea acelaşi format ca şi antetul funcţiei, în plus este urmat de punct şi virgulă. Pentru alte detalii vezi paragraful 1.11.

Limbajul C se livrează cu o serie de funcţii care au o utilizare frecventă în programe. Ele se păstrează într-un fişier special în format OBJ, adică compilat şi se adaugă la fiecare program în faza de editare. Aceste funcţii sînt numite *funcţii standard de bibliotecă*. Ele au prototipurile în diferite *fişiere de extensie .h*. Exemple de astfel de funcţii sînt funcţiile cu ajutorul cărora se realizează operaţii de intrare/ieşire (printf, scanf, puts, gets, getch, putch etc.), funcţiile pentru calculul funcţiilor elementare (sqrt, sin, cos, atan, log, pow etc.), funcţii de conversii (sscanf, sprintf etc.) etc.

Prototipurile funcţiilor standard se includ în program înaintea apelurilor lor folosind construcţia *#include* tratată prin preprocesor (vezi paragraful 1.12.).

Exemple de fişiere cu prototipuri:

stdio.h	pentru funcţiile printf, scanf, gets, puts etc.,
conio.h	pentru funcţiile putch, getch, getche etc.;
math.h	pentru funcţiile elementare sqrt, sin, cos etc.

Menţionăm că pînă în prezent în toate exerciţiile au fost apelate numai funcţii standard.

Apelul unei funcţii care nu returnează o valoare, se realizează printr-o *instrucţiune de apel*. Aceasta are formatul:

nume (*lista parametrilor efectivi*);

unde:

nume	- Este numele funcţiei care se apelează.
lista parametrilor efectivi	- Este fie vidă, cînd funcţia nu are parametri, fie o *expresie* sau mai multe separate prin *virgulă*.

Deci un *parametru efectiv* este o expresie.

Parametrii efectivi de la apel se corespund cu cei formali prin ordine. Primul parametru efectiv corespunde primului parametru formal, cel de al doilea parametru efectiv corespunde celui de al doilea parametru formal ş.a.m.d.

Amintim că parametrii unei funcţii (formali sau efectivi) se mai numesc şi *argumente*.

O funcţie care returnează o valoare, poate fi apelată fie printr-o *instrucţiune de apel*, fie ca *operand* al unei expresii. În cazul în care funcţia se apelează printr-o instrucţiune de apel, se pierde valoarea returnată. Cînd funcţia se apelează ca operand al unei expresii, valoarea returnată de ea se utilizează la evaluarea expresiei respective.

Spre exemplificare să considerăm funcţia *getch*. Noi am apelat această funcţie atît ca operand în expresii de atribuire de forma:

c=getch()

cît şi printr-o instrucţiune de apel de forma:

getch();

În primul caz, valoarea codului ASCII al caracterului citit de la tastatură se atribuie variabilei *c*. La cel de al doilea apel, valoare codului ASCII al caracterului citit de la tastatură nu este utilizată. În acest caz apelul se face cu scopul de a afişa ecranul utilizator şi de a bloca execuţia programului pînă la acţionarea unei taste oarecare, corespunzătoare caracterelor ASCII.

La apelul unei funcţii, valorile parametrilor efectivi se atribuie parametrilor formali corespunzători, apoi execuţia continuă cu prima instrucţiune din corpul funcţiei apelate.

În cazul în care tipul unui parametru efectiv diferă de tipul parametrului

formal care-i corespunde, în limbajul C se converteşte automat valoarea parametrului efectiv spre tipul parametrului formal respectiv. În limbajul C++ se utilizează o regulă mai complexă pentru apelul funcţiilor şi de aceea se recomandă să nu se folosească nici în limbajul C, regula de conversie automată amintită mai sus. În acest scop se poate utiliza operatorul de conversie explicită ((tip)), adică folosind aşa numitele expresii *cast*.

De exemplu, dacă funcţia *f* are un parametru de tip *double* şi *n* este o variabilă de tip *int*, atunci în locul apelului:

f(n);

se recomandă a folosi apelul cu conversie explicită a lui *n* spre tipul double:

f((double)n);

Acest al doilea apel se realizează identic în ambele limbaje.

La revenirea dintr-o funcţie apelată printr-o instrucţiune de apel se va continua cu execuţia instrucţiunii următoare celei care a făcut apelul. În cazul în care o funcţie este apelată ca un operand al unei expresii, la revenirea din ea se continuă cu evaluarea expresiei respective.

Revenirea dintr-o funcţie se poate realiza în unul din următoarele două moduri:

a. După execuţia ultimei instrucţiuni din corpul funcţiei (s-a ajuns la acolada închisă care termină corpul funcţiei).

b. La întîlnirea instrucţiunii *return*.

Instrucţiunea *return* este instrucţiunea de *revenire* dintr-o funcţie. Ea are formatele:

1. **return;**

   sau

2. **return** *expresie*;

Cel de al doilea format se foloseşte în corpul unei funcţii care returnează o valoare la revenirea din ea. Valoarea expresiei din instrucţiunea *return* este chiar valoarea returnată de funcţie.

În cazul în care tipul acestei expresii diferă de tipul care precede numele din antetul funcţiei, valoarea expresiei se converteşte automat spre tipul din antet, înainte de a se reveni din funcţie.

Formatul 1 al instrucţiunii de revenire se utilizează numai în corpul unei funcţii care nu returnează nici o valoare. Dintr-o astfel de funcţie se poate

reveni şi în modul indicat la punctul *a* de mai sus.

Faptul că instrucţiunea *return* (în ambele formate) poate fi scrisă în orice punct al corpului unei funcţii, permite o mai mare flexibilitate în programarea în limbajele C şi C++.

Exerciţii:

4.27 Să se scrie o funcţie care are ca parametru un întreg *n* din intervalul [0,170], calculează şi returnează pe n!

Numim *factorial* această funcţie. Ea returnează o valoare flotantă în dublă precizie. Rezultă că funcţia are antetul:

double factorial(int n)

La început funcţia verifică dacă *n* aparţine intervalului [0,170]. În cazul în care *n* nu aparţine acestui interval, funcţia va returna valoarea -1. Metoda de calcul este aceeaşi cu cea utilizată în exerciţiul 4.12.

FUNCŢIA BIV27

```
double factorial(int n)
/* calculeaza si returneaza pe n! pentru n in intervalul [0,170];
  altfel returneaza -1 */
{
 double f;
 int i;

 if( n < 0 || n > 170)
     return -1.0;
 for( i=2, f=1.0; i <= n; i++)
     f *= i;
 return f;
}
```

4.28 Să se scrie un program care calculează şi listează pe m! pentru

m=0, 1, 2, ..., 170.

Acest program apelează funcţia *factorial* definită mai sus pentru a-l calcula pe m! pentru o valoare dată a lui *m*.

Definiţia funcţiei *factorial* şi funcţia principală care o apelează pot fi editate în acelaşi fişier sursă. Cele două funcţii pot fi editate în orice ordine. În cazul în care definiţia funcţiei *factorial* se află în fişierul sursă după funcţia principală, apelul funcţiei *factorial* din funcţia principală, trebuie să fie precedat de prototipul ei.

PROGRAMUL BIV28

```
double factorial(int); /* prototipul functiei factorial */

#include <stdio.h>
#include <conio.h>

main() /* afiseaza pe m! pentru m = 0,1,2,...,170 */
{
 int m;

 for(m=0; m<171; m++) {
    printf("m=%d\tm!=%g\n",m,factorial(m));
    if((m+1)%23 == 0) {
        printf("actionati o tasta pentru a\
                continua\n");
        getch();
    } /* sfirsit if*/
  } /* sfirsit for */
} /* sfirsit main */

double factorial(int n)
/* calculeaza si returneaza n! pentru n in intervalul [0,170];
  altfel returneaza -1 */
{
 double f;
 int i;

 if( n < 0 || n > 170)
    return -1.0;
 for( i=2, f=1.0; i <= n; i++)
    f *= i;
 return f;
} /* sfirsit factorial */
```

Observaţii:

1. Funcţia *factorial* este apelată ca parametru efectiv în apelul funcţiei *printf*. La apelul funcţiei *factorial*, valoarea parametrului ei efectiv *m* se atribuie parametrului formal *n*. Această valoare este apoi folosită în corpul funcţiei factorial pentru a-l calcula pe n! şi deci şi pe m!. Valoarea respectivă se obţine ca valoare a lui *f*.

La revenirea din funcţie se returnează valoarea lui *f*, adică chiar *m!*, care se afişează folosind specificatorul de format *%g*.

2. Cele două funcţii pot fi editate în fişiere separate. De exemplu, presupunem că funcţia *factorial* se editează în fişierul BIV27.CPP, iar funcţia principală în fişierul BIV28A.CPP. În acest caz putem include fişierul BIV27.CPP folosind construcţia *#include* pentru a compila împreună cele două funcţii. Includerea se poate face înainte de definiţia funcţiei *main* sau după ea. În cazul în care facem includerea în faţa funcţiei *main*, nu mai este necesar să indicăm prototipul funcţiei *factorial* deoarece apelul ei este precedat chiar de definiţia funcţiei. Procedînd în acest fel se obţine varianta de mai jos.

PROGRAMUL BIV28A

```
#include <stdio.h>
#include <conio.h>
#include "biv27.cpp"

main() /* afiseaza pe m! pentru m = 0,1,2,...,170 */
{
 int m;

 for(m=0; m<171; m++) {
     printf("m=%d\tm!=%g\n",m,factorial(m));
     if((m+1)%23 == 0) {
         printf("actionati o tasta pentru a\
                continua\n");
         getch();
     }
 }
}
```

3. O altă posibilitate de compilare şi link-editare a funcţiilor unui program,editate în mai multe fişiere, este aceea de a utiliza un fişier de tip *Project* (cu extensia *.prj*). Un astfel de fişier, conţine numele fiecărui fişier (împreună cu extensia lui) care se compilează şi link-editează în vederea obţinerii fişierului executabil al programului. În acest fişier pot fi indicate nu numai fişiere de tip sursă (cu extensia .CPP), ci şi fişiere de tip obiect (cu extensia .OBJ) sau chiar fişiere cu biblioteci de funcţii, altele decît cele ale sistemului.

Fişierele de tip *project* pentru limbajele Turbo C şi C++ nu sînt compatibile. Ele se construiesc folosind meniul *Project* al celor două sisteme

intregate de dezvoltare.

Avantajele fişierelor de tip *Project* constau în aceea că, la fiecare lansare se compilează în mod automat numai sursele în care s-au făcut modificări. De aceea, în cazul programelor sursă mari se recomandă împărţirea lor în mai multe fişiere sursă şi compilarea lor prin utilizarea fişierelor de tip *Project*. De obicei, într-un fişier sursă se grupează funcţii "înrudite", adică funcţii care prelucrează în comun subseturi de date sau sînt logic legate între ele.

Întrucît exerciţiile pe care le prezentăm nu sînt de dimensiuni mari, vom utiliza frecvent includerile de fişiere folosind construcţia ***#include*** a preprocesorului.

4.29 Să se scrie o funcţie care are ca parametri doi întregi *x* şi *y*, calculează şi returnează numărul aranjamentelor de *x* obiecte luate cîte *y*.

La început funcţia testează dacă *x* aparţine intervalului [1,170], iar *y* intervalului [1,x]. În caz de eroare, funcţia returnează valoarea -1.

Dacă notăm cu A(x,y) numărul aranjamentelor de *x* obiecte luate cîte *y*, atunci:

A(x,y)=x*(x-1)*(x-2)*...*(x-y+1)

FUNCŢIA BIV29

```
double aranjamente ( int x, int y)
/* calculeaza si returneaza numarul aranjamentelor de x obiecte luate
   cite y */
{
 double a;
 int i;

 if(x < 1 || x > 170 )
    return -1.0;
 if(y < 1 || y > x )
    return -1.0;
 a = 1.0;
 i = x - y + 1;
 while( i <= x )
    a *= i++;
 return a;
}
```

4.30 Să se scrie un program care calculează şi afişează numărul aranjamentelor de *n* obiecte luate cîte *k*, pentru n=1,2,...,170 şi k=1,2,...,n.

Programul de faţă apelează funcţia *aranjamente* definită în exerciţiul precedent.

PROGRAMUL BIV30

```
#include <stdio.h>
#include <conio.h>
#include "biv29.cpp" /* contine functia aranjamente */

main() /* calculeaza si afiseaza numarul aranjamentelor de n obiecte
          luate cite k, pentru n in intervalul [1,170] si k in
          intervalul [1,n] */
{
 int k,n;

 for(n = 1; n <= 170; n++) {
     printf("actionati o tasta pentru a\
           continua\n");
     getch();
     printf("\nn= %d\n",n);
     for(k=1; k <= n; k++) {
         printf("k=%d\tA(n,k)=%g\n",k,
               aranjamente(n,k));
         if(k%23 == 0 ) {
             printf("actionati o tasta pentru a\
                   continua\n");
             getch();
         } /* sfirsit if*/
     } /* sfirsit for interior */
   } /* sfirsit for exterior */
} /* sfirsit main*/
```

4.31 Să se scrie o funcţie care are ca parametri doi întregi *x* şi *y*, calculează şi returnează numărul combinărilor de *x* elemente luate cîte *y*.

Funcţia testează dacă *x* aparţine intervalului [1,170], iar *y* intervalului [0,x]. În caz de eroare, funcţia returnează valoarea -1.

Pentru calculul numărului combinărilor de *x* elemente luate cîte *y* se determină raportul dintre numărul aranjamentelor de *x* elemente luate cîte *y* şi y!.

În acest scop, funcţia de faţă apelează funcţiile *aranjamente* şi *factorial* definite în exerciţiile 4.29. şi respectiv 4.27.

FUNCŢIA BIV31

```
#include "biv27.cpp"
#include "biv29.cpp"

double combinari(int x, int y )
/* calculeaza si returneaza numarul combinarilor de x obiecte
   luate cite y */
{
 if( x < 1 || x > 170 )
    return -1.0;
 if( y < 0 || y > x )
    return -1.0;
 if( y == 0 || y == x )
    return 1.0;
 return aranjamente(x,y)/factorial(y);
}
```

Observaţii:

1. Un calcul mai rapid se realizează dacă se ţine seama de relaţia

   c(x,y)=c(x,x-y)

   unde prin c(x,y) s-a notat numărul combinărilor de *x* obiecte luate cîte y. Această relaţie este util să se aplice pentru

   y > x/2.

   În acest scop ultima instrucţiune se înlocuieşte cu:

```
if(y > x/2)
 return aranjamente (x,x-y)/factorial(x-y);
else
 return aranjamente (x,y)/factorial(y);
```

2. O altă variantă pentru calculul numărului combinărilor este relaţia:

   c(x,y)=(x/1)*((x-1)/2)*((x-2)/3)*...*((x-y+1)/y)

   În acest caz se evită calculul valorilor A(x,y) şi y! care cresc rapid odată cu creşterea valorilor lui *x* şi *y*. Aceasta se realizează cu ajutorul funcţiei de mai jos.

FUNCŢIA BIV31A

```
double combinari(int x, int y )
/* calculeaza si returneaza numarul combinarilor de x obiecte
   luate cite y */
{
 int i;
 double c;

 if( x < 1 || x > 170 )
    return -1.0;
 if( y < 0 || y > x )
    return -1.0;
 if( y == 0 || y == x )
    return 1.0;
 c = 1.0;
 for( i = 1; i <= y; i++)
    c = (c * (x - i + 1)) / i;
 return c;
}
```

4.32 Să se scrie un program care calculează şi afişează numărul combinărilor de *n* obiecte luate cîte *k*, pentru n=1,2,...,170 şi k=0,1,2,...,n.

Programul utilizează funcţia BIV31A.

PROGRAMUL BIV32

```
#include <stdio.h>
#include <conio.h>
#include "biv31a.cpp"

main() /* afiseaza numarul combinarilor de n obiecte luate cite k
          pentru n in intervalul [1,170] si k in intervalul [0,n] */
{
 int k,n;

 for(n=1; n<= 170; n++) {
    printf("actionati o tasta pentru a\
          continua\n");
    getch();
    printf("\nn=%d\n",n);
    for( k=0; k <=n ; k++) {
       printf("k=%d\tC(n,k)=%g\n", k,
```

```
            combinari(n,k));
         if((k+1)%23 == 0) {
            printf("actionati o tasta pentru a\
                  continua\n");
            getch();
         } /* sfirsit if */
      } /* sfirsit for interior */
   } /* sfirsit for exterior */
} /* sfirsit main */
```

4.33 Să se scrie o funcţie care ridică la o putere întreagă nenegativă un număr flotant de tip *long double*.

Funcţia utilizează metoda expusă în exerciţiul 4.19.

FUNCŢIA BIV33

```
long double ldrp( long double x, int n)
/* ridica pe x la puterea n */
{
 long double f,p;

 for(p=x,f=1.0L; n ; n >>= 1) {
     if( n&1)
        f *= p;
     if( n > 1)
        p *= p;
 }
 return f;
}
```

4.34 Să se scrie un program care tabelează puterile întregi ale unui număr flotant de tip *long double*, exponentul variind, cu pasul 1, între două limite *m* şi *n*.

PROGRAMUL BIV34

```
#include <stdio.h>
#include <stdlib.h>
#include "biv33.cpp"

main() /* tabeleaza puterile intregi ale unui numar*/
{
```

```
char t[255];
long double f,g;
int i,m,n;

do {
   printf("tastati numarul care se ridica la\
        puteri: ");
   if( gets(t) == NULL ) {
      printf("s-a tastat EOF\n");
      exit(1);
   }
   if(sscanf(t,"%Lf",&f) == 1 )
      break;
   printf("nu s-a tastat un numar\n");
} while(1);
do {
   do {
      printf("tastati limita inferioara a\
           exponentului: ");
      if(gets(t) == NULL ) {
         printf("s-a tastat EOF\n");
         exit(1);
      }
      if(sscanf(t,"%d",&m) == 1)
         break;
      printf("nu s-a tastat un intreg\n");
   } while (1);

   do {
      printf("tastati limita superioara a\
           exponentului: ");
      if(gets(t) == NULL ) {
         printf("s-a tastat EOF\n");
         exit(1);
      }
      if(sscanf(t,"%d", &n ) == 1 )
         break;
      printf("nu s-a tastat un intreg\n");
   } while(1);

   if(m <= n)
      break;
   printf("limita inferioara:%d o depaseste\
```

```
            pe cea superioara:%d\n", m,n);
   } while (1);
   if( f == 0 && m < 0 ) {
      printf("putere negativa pentru zero\n");
      exit(1);
   }
   for( i=m; i <= n; i++) {
      if( i < 0 )
         g = 1.0/ldrp(f,-i);
      else
         g = ldrp(f,i);
      printf("%Lg la puterea %d este: %Lg\n",f,i,g);
   }
}
```

4.15. Apel prin valoare şi apel prin referinţă

În limbajul C, la apelul unei funcţii, fiecărui parametru formal i se atribuie *valoarea* parametrului efectiv care-i corespunde. Deci, la apelul unei funcţii se transferă valorile parametrilor efectivi. Din această cauză, se spune că apelul este prin *valoare* (*call by value*).

Exemplu:

Fie funcţia *factorial* de prototip:

double *factorial*(**int** *n*);

definită în exerciţiul 4.27. La apelul:

fact=factorial(10);

se atribuie parametrului formal *n* valoarea 10. Această atribuire se face înainte de execuţia primei instrucţiuni a funcţiei *factorial*. De aceea, prin acest apel funcţia factorial calculează pe 10!.

În unele limbaje de programare, la apel nu se transferă valorile parametrilor efectivi, ci *adresele* acestor valori. În acest caz, se spune că apelul este prin *referinţă* (*call by reference*).

Între cele două tipuri de apeluri există o diferenţă esenţială şi anume:

- în cazul apelului prin valoare, funcţia apelată nu poate modifica parametri efectivi din funcţia care a făcut apelul, neavînd acces la ei;
- în cazul apelului prin referinţă, funcţia apelată, dispunînd de adresa parametrilor efectivi, îi poate modifica.

Aşa se întîmplă, de exemplu, în cazul limbajului FORTRAN, unde

apelul este prin referinţă.

De aici, rezultă că la apelul prin valoare transferul datelor este *unidirecţional*, adică valorile se transferă prin parametri numai de la funcţia care face apelul spre cea care a fost apelată, nu şi invers.

Există situaţii cînd se doreşte ca funcţia apelată să modifice valorile parametrilor efectivi. Unele limbaje permit ambele tipuri de apeluri. Aşa este, de exemplu, cazul limbajului Pascal. Parametri declaraţi în Pascal prin *var* se transferă prin referinţă.

În limbajul C++, spre deosebire de limbajul C, există ambele apeluri, ca şi în limbajul Pascal.

Apelul prin valoare este util în cazul în care funcţia apelată nu trebuie să modifice valorile parametrilor efectivi, deoarece în felul acesta parametrii efectivi sînt protejaţi faţă de eventualele modificări care ar putea apare din greşeală.

Aceasta înseamnă că o funcţie poate să modifice parametrii ei formali fără riscul ca prin modificarea respectivă să influenţeze valorile parametrilor efectivi corespunzători. Aşa de exemplu, funcţia *ldrp*, definită în exerciţiul 4.33, modifică valoarea parametrului formal *n* la zero. Cu toate acestea, valoarea parametrului efectiv corespunzător lui *n* nu suferă nici o modificare, avînd aceeaşi valoare la revenirea din funcţie ca şi în momentul apelului.

În limbajul C, un parametru efectiv poate fi numele unui tablou. De exemplu, fie o funcţie, pe care o numim *sum* şi care însumează elementele unui tablou de tip *int*. Această funcţie are doi parametri:

tab - Este numele tabloului ale cărui elemente se însumează.

n - Este numărul elementelor tabloului.

Ea returnează suma:

tab[0]+tab[1]+...+tab[n].

Conform celor spuse mai sus, funcţia are următorul antet:

int sum(int tab[],int n)

Amintim că dacă un parametru efectiv este un nume de tablou unidimensional, atunci parametrul formal corespunzător se poate declara fără a indica, în parantezele pătrate, limita superioară a indicelui său.

Dacă *a* este un tablou declarat ca mai jos:

int a[10];

atunci pentru a însuma elementele tabloului respectiv vom putea folosi

apelul:

x=sum(a,10);

Deoarece *a* este numele unui tablou, *a* are ca valoare adresa primului său element, adică adresa lui a[0]. Apelul fiind prin valoare, parametrul *tab* primeşte valoarea lui *a* deci *tab* are şi el ca valoare adresa lui a[0]. Rezultă că tab[0], tab[1], ..., tab[9] reprezintă aceleaşi elemente ca a[0], a[1], ..., a[9]. De aceea, corpul funcţiei *sum* referindu-se la elementele lui *tab*, se referă de fapt chiar la elementele lui *a*:

```
for(s=0;n>0;n--)
 s+=tab[n-1];
```

Se observă că, în acest caz, deşi apelul s-a făcut prin valoare (valoarea lui *a* s-a atribuit lui *tab*), s-a realizat un apel prin referinţă, deoarece valoarea atribuită parametrului formal este o adresă.

În acest caz, funcţia apelată poate modifica elementele tabloului al cărui nume s-a folosit ca parametru efectiv.

Într-adevăr, în exemplul de mai sus, atribuirea:

tab[3]=100;

scrisă în corpul funcţiei *sum* are ca efect atribuirea valorii 100 elementului a[3].

În concluzie, în limbajul C, apelul este prin *valoare*. Acest apel *devine* prin *referinţă* în cazul în care parametrul efectiv este numele unui tablou.

Ulterior o să vedem că apelul prin referinţă poate fi realizat în limbajul C şi în cazul variabilelor simple, folosind *pointeri*.

De asemenea, într-un alt capitol o să vedem că în limbajul C++ putem să stabilim, pentru fiecare parametru, tipul de apel: prin *valoare* sau prin *referinţă*.

Exerciţii:

4.35 Să se scrie o funcţie care calculează valoarea unui polinom folosind schema lui Horner.

Parametrii funcţiei sînt:

c	- Tabloul cu coeficienţii polinomului.
	- Este de tip *long double*.
n	- Gradul polinomului.
	- Este de tip *int*.
x	- Valoarea variabilei.

- Este de tip *long double*.

Tabloul *c* conţine coeficienţii polinomului în ordine descrescătoare a puterilor lui *x*.

c[0]	- Este coeficientul lui x**n.
c[1]	- Este coeficientul lui x**(n-1).
...	-
c[n-1]	- Este coeficientul lui x.
c[n]	- Este termenul liber.

Se ştie că un polinom *p(x)*, cu coeficienţii c[0], c[1], ..., c[n], în ordine descrescătoare ale puterilor lui *x*, se poate pune sub forma:

p(x)=((...((c[0]*x+c[1])*x+c[2])*x+...+c[n-2])*x+c[n-1])*x+c[n]

Fie

s=c[0]

Se observă că paranteza cea mai interioară are ca valoare, valoarea expresiei:

s*x+c[1]

Să atribuim lui *s* valoarea acestei expresii:

s = s*x+c[1]

În continuare trebuie să se evalueze valoarea expresiei:

s*x+c[2]

Atribuim din nou lui *s* valoarea acestei expresii:

s=s*x+c[2]

În general, la al *i*-lea pas se evaluează expresia

s*x+c[i]

şi apoi valoarea ei se atribuie lui *s*:

s=s*x+c[i]

Deci, calculul valorii polinomului *p(x)* revine la execuţia repetată a atribuirii

s=s*x+c[i], pentru i=1, 2, ..., n,

valoarea iniţială a lui *s* fiind c[0]. Aceasta conduce la instrucţiunea *for* de mai jos:

```
for(s=c[0],i=1;i<=n;i++)
 s=s*x+c[i];
```

FUNCŢIA BIV35

```
long double polinom(long double x, int n,
      long double c[] )
/* calculeaza valoarea polinomului
   p(x) = c[0]*x**n + c[1]*x**(n-1) +...+ c[n-1]*x + c[n] */
{
 long double s;
 int i;

 for(s = c[0], i = 1; i <= n; i++ )
           s = s*x + c[i];
 return s;
}
```

Observaţie:

Biblioteca standard a limbajului C conţine funcţia *poly* care calculează valoarea unui polinom de gradul *n* cu coeficienţii de tip *double*. Ea are prototipul:

double poly(double *x*, int *n*, double *c[]*);

care se află în fişierul *math.h*.

În acest caz, c[0] este termenul liber, c[1] coeficientul lui *x*. În general c[i] este coeficientul lui *x* la puterea *i*.

4.36 Să se scrie un program care realizează următoarele:

- citeşte pe *x* de tip *long double*;
- citeşte pe *n* de tip *int* din intervalul [0,100];
- citeşte coficienţii a[0], a[1], ..., a[n] ai polinomului *p(x)* de tip *long double*;
- calculează şi afişează valoarea polinomului *p(x)*.

a[0] este coeficientul lui x**n, a[1] este coeficientul lui x**(n-1) etc.

PROGRAMUL BIV36

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include "biv35.cpp"

#define MAXGRAD 100
main() /* citeste pe x, n, a[0], a[1], ..., a[n], calculeaza si
          afiseaza valoarea polinomului
       p(x)=a[0]*x**n + a[1]*x**(n-1) +...+a[n-1]*x + a[n] */
{
 long double x,a[MAXGRAD+1];
 int i,n;
 char t[255];

/* citeste valoarea lui x */
 do {
     printf("valoarea lui x: ") ;
     if(gets(t) == NULL ) {
         printf("s-a tastat EOF\n");
         exit(1);
     }
     if(sscanf(t,"%Lf", &x) == 1 )
         break;
     printf("nu s-a tastat un numar\n");
 } while(1);

/* citeste pe n */
 do {
     printf("valoarea lui n: ");
     if(gets(t) == NULL ) {
         printf("s-a tastat EOF\n");
         exit(1);
     }
     if(sscanf(t,"%d", &n) == 1 && n >= 0 &&
           n <= 100 )
         break;
     printf("nu s-a tastat un intreg din\
            intervalul [0,100]\n");
 } while (1);

/* citeste coeficientii polinomului */
 for( i=0; i <= n; i++ )
     do {
         printf("coeficientul a[%d]: ",i);
         if(gets(t) == NULL ) {
             printf("s-a tastat EOF\n");
```

```
            exit(1);
        }
        if(sscanf(t,"%Lf", &a[i]) == 1)
            break; /* iesire din do - while */
        printf("nu s-a tastat un numar\n");
    } while(1);
/* sfirsit citire coeficienti polinom */

 printf("x=%Lg\tgrad=%d\tval pol=%Lg\n",x,n,
        polinom(x,n,a));
}
```

4.37 Să se scrie o funcţie care citeşte cel mult *n* numere şi le păstrează în tabloul *ndtab* care este de tip *double*. Funcţia returnează numărul numerelor citite.

FUNCŢIA BIV37

```
int ndcit(int n, double ndtab[] )
/* citeste cel mult n numere si le pastreaza in tabloul ndtab;
   returneaza numarul numerelor citite */
{
 int i;
 double d;
 char t[255];

 for(i=0; i < n; i++) {
     printf("elementul [%d]= ",i);
     if(gets(t) == NULL )
         return i;
     if(sscanf(t,"%lf", &d) != 1)
         break;
     ndtab[i] = d;
 }
 return i;
}
```

4.38 Să se scrie o funcţie care citeşte componentele unui vector precedate de numărul lor. Funcţia returnează numărul componentelor vectorului respectiv.

Această funcţie foloseşte funcţia definită în exerciţiul precedent.

FUNCŢIA BIV38

```
int dvcit(int nmax, double dvector[])
/* citeste un intreg n si cele n componente ale vectorului
  dvector; returneaza numarul componentelor citite */
{
 int i,n;
 char t[255];

 do {
     printf("numarul elementelor= ");
     if(gets(t) == NULL ) {
         printf("s-a tastat EOF\n");
         exit(1);
     }
     if(sscanf(t,"%d", &n) == 1 && n > 0 &&
         n <= nmax )
         break;
     printf("nu s-a tastat un intreg din\
             intervalul [1,%d]\n", nmax);
 } while(1);
 if((i=ndcit(n,dvector)) != n ) {
     printf("nu s-a reusit citirea celor n=%d\
             componente\n",n);
     printf("s-au citit numai %d componente\n",i);
 }
 return i;
}
```

4.39 Să se scrie un program care citeşte componentele vectorilor *x* şi *y*, calculează şi afişează produsul lor scalar.

Componentele celor doi vectori se tastează astfel:

- numărul componentelor vectorilor (*n*);
- cele *n* componente ale lui *x*;
- cele *n* componente ale lui *y*.

Programul de faţă utilizează funcţia definită în exerciţiul precedent.

PROGRAMUL BIV39

```
#include <stdio.h>
#include <stdlib.h>
#include "biv37.cpp"
```

```
#include "biv38.cpp"

#define MAX 1000

main() /* citeste componentele vectorilor x si y precedate de
           numarul lor; calculeaza si afiseaza produsul lor scalar */
{
 int i,n;
 double x[MAX],y[MAX],s;

/* citeste pe n si componentele vectorului x */
 n = dvcit(MAX,x);

/* citeste componentele vectorului y */
 if(ndcit(n,y) != n ) {
     printf("nu sint n=%d componente pentru\
            y\n",n);
     exit(1);
 }

/* calculeaza produsul scalar */
 s = 0.0;
 for(i=0; i < n; i++)
     s += x[i]*y[i];
 printf("(x,y)= %g\n", s);
}
```

4.40 Să se scrie o funcţie care sortează în ordine crescătoare, elementele unui tablou de tip *double*.

Spunem despre un tablou *tab*, de elemente numerice, că este sortat crescător (descrescător), dacă elementele lui satisfac relaţia:

tab[i] <= (>=) tab[i+1]

pentru

i=0,1,2,...,n-2

unde:

n - Este numărul elementelor tabloului *tab*.

Funcţia care realizează sortarea are doi parametrii:

tab - Tabloul de sortat.

n - Numărul elementelor tabloului.

La ora actuală, există o serie de metode de sortare, care diferă între ele, mai ales, prin ordinul de mărime al paşilor necesari pentru a realiza sortarea. Pentru amănunte se poate consulta [3].

Cele mai simple metode sînt cele mai puţin eficiente şi necesită un număr de paşi de ordinul lui n*n, *n* fiind numărul elementelor de sortat. Prezentăm mai jos o astfel de metodă, care este cunoscută sub denumirea de *metoda bulelor* (*bubble* sort).

Ea constă în următoarele:

Se compară primele două elemente ale tabloului de sortat şi dacă nu sînt în ordinea cerută (de exemplu crescătoare), se permută între ele. Apoi se compară elementul al doilea cu al treilea şi se procedează ca mai sus. În general, se compară două elemente învecinate, să zicem al *i*-lea cu al *i+1*-lea şi dacă nu sînt în ordinea cerută, atunci ele se *permută*. Se procedează în acest fel cu toate perechile de elemente învecinate ale tabloului de sortat. Apoi procesul respectiv se reia de la început. El se întrerupe cînd se constată că toate elementele învecinate ale şirului sînt în ordinea cerută (în cazul de faţă, crescătoare).

Exemplu:

Fie şirul de numere:

8,3,9,5,4,7

pe care dorim să-l ordonăm crescător.

Prima parcurgere a şirului:

8 3 9 5 4 7 şirul iniţial
3 8 9 5 4 7 şirul după permutarea lui 8 cu 3
3 8 5 9 4 7 şirul după permutarea lui 9 cu 5
3 8 5 4 9 7 şirul după permutarea lui 9 cu 4
3 8 5 4 7 9 şirul după permutarea lui 9 cu 7.

A doua parcurgere a şirului:

3 8 5 4 7 9 şirul după prima parcurgere
3 5 8 4 7 9 şirul după permutarea lui 8 cu 5
3 5 4 8 7 9 şirul după permutarea lui 8 cu 4
3 5 4 7 8 9 şirul după permutarea lui 8 cu 7.

A treia parcurgere a şirului:

3 5 4 7 8 9 şirul după a doua parcurgere a lui
3 4 5 7 8 9 şirul după permutarea lui 5 cu 4.

La a patra parcurgere a şirului nu se mai face nici o permutare deoarece

perechile de elemente vecine sînt în ordine crescătoare, deci şirul este sortat crescător.

O proprietate a acestei metode este aceea că la fiecare parcurgere a şirului de numere, cel puţin un element ajunge să-şi ocupe poziţia sa finală în conformitate cu ordonarea avută în vedere. Astfel, după prima parcurgere a şirului, elementul ajuns pe ultimul loc nu va mai fi permutat, ocupîndu-şi locul final. În exemplul de mai sus,9 a ajuns pe ultimul loc după prima permutare, poziţie care este finală pentru el. După cea de a doua parcurgere, 8 a ajuns şi el pe poziţia lui finală. Întîmplător şi 7 a ajuns pe poziţia lui finală la cea de a doua parcurgere şi aşa mai departe.

Dacă se ordonează crescător un tablou de numere, atunci după prima parcurgere a elementelor lui, elementul maxim va avea indicele cel mai mare. După cea de-a doua parcurgere, maximul dintre elementele rămase (fără a considera ultimul element) va ocupa penultima poziţie în tablou etc.

Se obişnuieşte să se spună că elementele urcă în poziţiile lor la fel ca şi bulele de aer în apa care fierbe, de unde şi denumirea metodei.

Funcţia de mai jos conţine un ciclu pentru parcurgerea elementelor tabloului *tab*. În acest ciclu se compară elementele vecine curente şi dacă acestea nu sînt în ordine crescătoare, ele se permută. Deoarece, după o parcurgere a şirului se reîncepe parcurgerea lui, acest ciclu este conţinut într-un altul care realizează execuţia repetată a ciclului de parcurgere a şirului. Se ajunge în acest fel la următorii paşi:

1. Cît timp şirul nu este sortat se execută pasul 2.

   Altfel se întrerupe procesul de sortare.

2. Se parcurge şirul şi se permută perechile de elemente învecinate care sînt în ordine descrescătoare.

Pasul 2 se realizează printr-un ciclu *for* de forma:

```
for(i=0;i<n-1;i++)
 if(tab[i]>tab[i+1]){

/* se permută elementele tab[i] şi tab [i+1] */
     t=tab[i];
     tab[i]=tab[i+1];
     tab[i+1]=t;
 }
```

Acest ciclu trebuie repetat atît timp cît elementele tabloului *tab* nu sînt în ordine crescătoare. Problema care se pune în legătură cu execuţia repetată a acestui ciclu *for* este aceea de a şti cînd s-a ajuns la situaţia în care

elementele tabloului *tab* sînt în ordine crescătoare. Observăm că în acest caz ciclul *for* se execută fără a mai face permutări. Deci, tabloul este sortat cînd ciclul *for* îl parcurge fără a face permutări. Acest fapt poate fi detectat folosind o variabilă, să zicem *ind*, care se anulează înaintea fiecărei execuţii a ciclului *for* şi se setează la valoarea 1 în cazul în care ciclul respectiv face o permutare. În felul acesta, variabila *ind* are valoarea 1 după execuţia ciclului *for*, dacă acesta a făcut cel puţin o permutare şi zero în caz contrar.

În concluzie, variabila *ind* are valoare zero cînd şirul este sortat şi 1 în caz contrar. Variabila *ind* se gestionează astfel:

- iniţial *ind* are valoarea 1;
- înainte de instrucţiunea *for*, *ind* se face egal cu zero;
- în secvenţa de permutare a două elemente învecinate, *ind* se face egal cu 1.

Se obţine secvenţa:

```
ind = 0;
for (i=0;i<=n-1;i++)
 if(tab[i]>tab[i+1]){
     t=tab[i];
     tab[i]=tab[i+1];
     tab[i+1]=t;
     ind=1;
 }
```

Această secvenţă se execută repetat atît timp cît *ind* nu este egal cu zero, adică tabloul nu este sortat. În acest scop se foloseşte un ciclu *while* care are ca şi corp chiar secvenţa de mai sus:

```
while(ind){
 ind=0;
 for(i=0;i<=n-1;i++)
     if(tab[i]>tab[i+1]){
         ...
     } /* sfirsit if */
} /* sfirsit while */
```

FUNCŢIA BIV40

```
void ordcresc(double tab[], int n)
/* ordoneaza elementele lui tab in ordine crescatoare */
{
 int i,ind;
 double t;
```

```
    ind = 1;
    while(ind) {
        ind = 0;
        for(i=0; i< n-1; i++)
            if(tab[i] > tab[i+1]) {
                t = tab[i];
                tab[i] = tab[i+1];
                tab[i+1] = t;
                ind = 1;
            } /* sfirsit if */
    } /* sfirsit while */
}
```

4.41 Să se scrie un program care citeşte un şir de numere separate prin caractere albe şi le afişează în ordine crescătoare. După ultimul număr se tastează un caracter nenumeric. Se presupune că la intrare există cel puţin două numere şi cel mult 1000.

Numerele se citesc apelînd funcţia *ndcit* definită în exerciţiul 4.37. Sortarea lor în ordine crescătoare se realizează folosind funcţia *ordcresc* definită în exerciţiul precedent.

PROGRAMUL BIV41

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include "biv37.cpp"
#include "biv40.cpp"

#define MAX 1000

main() /* citeste un sir de numere si le afiseaza in ordine crescatoare */
{
 double tnr[MAX];
 int n,i;

 if((n=ndcit(MAX,tnr)) < 2) {
    printf("s-au citit mai putin de 2 numere\n");
    exit(1);
 }
 ordcresc(tnr,n);
 for(i=0; i < n; i++) {
```

```
        printf("tnr[%d]=%g\n", i,tnr[i]);
        if((i+1)%23 == 0) {
            printf("actionati o tasta pentru a\
                    continua\n");
            getch();
        }
    }
}
```

4.42 Să se scrie o funcţie care caută un element de valoare dată *a*, într-un tablou de numere sortat crescător. Funcţia returnează indicele unui element de valoare egală cu *a* sau -1 în cazul în care nu există un astfel de element.

Elementul de valoare *a* poate fi găsit comparînd fiecare element al tabloului cu *a* pînă cînd se găseşte un element egal cu *a*. În cazul în care nu există un astfel de element în tabloul respectiv, compararea se face cu toate elementele tabloului pentru a putea constata că nu există elementul căutat. O astfel de metodă de căutare, numită căutare *liniară*, deşi este foarte simplă, ea necesită un număr relativ mare de comparări.

În cazul tablourilor sortate se poate utiliza o metodă de căutare mult mai rapidă, numită metoda *căutării binare*. Această metodă constă în următoarele.

Fie *tab* tabloul în care se caută un element de valoare *a* şi care este sortat crescător. Deci:

tab[0] <= tab[1] <= ... <= tab[n-1]

1. Se compară elementul din mijlocul tabloului (de indice [(n-1)/2]) cu *a*.

2. Dacă acesta are valoarea egală cu *a*, înseamnă că s-a găsit un element de valoare *a* şi se returnează indicele lui.

   Altfel se continuă cu punctul 3.

3. Dacă elementul din mijlocul tabloului este diferit de *a*, atunci din cauză că tabloul este sortat, elementul căutat se află fie în prima jumătate a tabloului, fie în a doua jumătate.

   Mai mult decît atît, se poate stabili exact în care jumătate urmează a se face căutarea şi anume:

   - Dacă tab[(n-1)/2]>a, atunci elementul căutat se poate afla numai în prima jumătate a tabloului *tab*. Deoarece elementele tabloului următoare lui tab[(n-1)/2], sînt mai mari decît el, deci urmează a face căutarea în subtabloul:

tab[0]<=tab[1]<=...<=tab[(n-1)/2-1].

- Dacă tab[(n-1)/2]<a, atunci elementul căutat se află numai în jumătatea a doua a tabloului *tab*. Deoarece elementele tabloului, precedente lui tab[(n-1)/2], sînt mai mici decît el, deci căutarea se va face în subtabloul:

tab[(n-1)/2+1]<=tab[(n-1)/2+2]<=...<=tab[n-1].

4. După determinarea subtabloului în care trebuie căutat elementul de valoare *a*, se procedează în mod analog, adică se determină elementul aflat la mijlocul acestui subtablou şi apoi se procedează ca la paşii 2 şi 3 de mai sus şi aşa mai departe.

Exemplu:

Fie tabloul *tab* de 10 elemente care au valorile:

tab[0]=3	tab[1]=4	tab[2]=7
tab[3]=10	tab[4]=13	tab[5]=20
tab[6]=21	tab[7]=35	tab[8]=41
tab[9]=48		

Se cere indicele elementului de valoare 20.

Avem:

n=10, [(n-1)/2]=[9/2]=4.

Se compară tab[4] cu 20:

tab[4]=13<20

deci pentru căutarea următoare se alege jumătatea a doua a tabloului:

tab[5]=20, tab[6]=21, tab[7]=35, tab[8]=41, tab[9]=48.

Elementul din mijlocul acestui tablou este de indice [(5+9)/2]=7. Avem:

tab[7]=35>20

deci se va face căutarea în prima jumătate a acestui subtablou, adică se selectează subtabloul:

tab[5]=20, tab[6]=21

În continuare, se alege elementul de indice [(5+6)/2]=5, deci tab[5]. Cum tab[5]=20, rezultă că în acest moment a fost găsit elementul căutat şi se revine din funcţie cu valoarea 5 (indicele elementului căutat).

Pentru determinarea, la fiecare pas, a subtabloului în care să se facă căutarea este bine să utilizăm două variabile *inf* şi *sup*, prima avînd ca valoare indicele primului element al subtabloului curent selectat, iar cea

de-a doua, indicele ultimului element al aceluiaşi subtablou. Paşii procesului de căutare vor fi:

1. inf=0

   Indicele primului element al tabloului.

2. sup=n-1

   Indicele ultimului element al tabloului.

3. i = (inf+sup)/2

   Indicele elementului din mijlocul tabloului.

4. Dacă

   tab[i]=a

   căutarea se termină şi se returnează valoarea lui *i*.

   Altfel se continuă cu 5.

5. Dacă

   tab[i] > a

   atunci

   sup=i-1

   Limita superioară se coboară la valoarea i-1 pentru a selecta prima jumătate a tabloului.

   Se reia de la punctul 3.

   Altfel înseamnă că tab[i] < a şi atunci

   inf=i+1

   Limita inferioară creşte la valoarea i+1 pentru a selecta jumătatea a doua a tabloului.

   Se reia de la punctul 3.

Acest proces se desfăşoară corect în cazul în care, în tablou, există cel puţin un element de valoare *a*. În caz contrar, el continuă indefinit. Se observă că la pasul 5 de mai sus, una din limite se modifică şi anume, dacă se modifică limita *sup*, atunci aceasta *descreşte*, iar dacă se modifică limita *inf*, atunci aceasta *creşte*. Iniţial inf <= sup.

Deoarece la fiecare pas la care nu s-a găsit elementul căutat, una dintre variabilele *inf* sau *sup* se modifică şi *inf* creşte, iar *sup* descreşte, rezultă că inegalitatea de mai sus nu va mai fi satisfăcută după un număr finit de paşi,

adică se ajunge ca să fie satisfăcută relaţia:

inf > sup

Aceasta se întîmplă cînd în tablou nu există un element de valoare *a*. Deci procesul de calcul indicat mai sus se întrerupe fie la găsirea elementului căutat, fie cînd se ajunge ca *inf* să-l depăşească pe *sup*.

Să reluăm exemplul de mai sus pentru a=22. Şi în acest caz se va ajunge la subtabloul:

tab[5]=20, tab[6]=21

şi

inf=5, sup=6

La pasul 3 se determină:

i=[(5+6)/2]=5

Apoi se ajunge la pasul 5, deoarece

tab[5]!=22.

Cum tab[5]=20 < 22, rezultă că inf=i+1=5+1=6. Se revine la pasul 3 unde se atribuie lui *i* valoarea:

i=[(inf+sup)/2]=[(6+6)/2]=6

Se ajunge din nou la pasul 5, deoarece

tab[6] != 22.

Cum tab[6]=21 < 22, se modifică din nou *inf*:

inf=i+1=7

În acest moment inf > sup şi deci procesul de calcul se întrerupe, concluzia fiind aceea că tabloul respectiv nu conţine un element de valoare *a*.

FUNCŢIA BIV42

```
int dcautbin(double tab[], int n, double a)
/* cauta in tab un element de valoare egala cu a;
  returneaza indicele elementului respectiv sau -1, daca nu exista un
   astfel de element */
{
 int i,inf,sup;

 inf = 0;
```

```
    sup = n-1;
    while(inf <= sup ) {
        i =(inf+sup)/2;
        if(tab[i] == a)
            return i;
        if(tab[i] > a)
            sup = i-1;
        else
            inf = i+1;
    } /* sfirsit while */

    /* se ajunge aici cind inf > sup, ceea ce inseamna ca in tab nu exista
     un element de valoare a */
    return -1;
}
```

4.43 Să se scrie un program care citeşte un şir de întregi separaţi prin caractere albe şi-i afişează pe cei care sînt numere prime şi sînt în intervalul (0,1000).

După ultimul număr se tastează un caracter nenumeric.

Acest program generează la început un tablou cu numere prime. În acest scop se realizează o funcţie pe care o numim *prim*. Ea are doi parametri: *tprim* şi *n*. Primul parametru este un tablou de tip *int*, ale cărui elemente sînt numerele prime care nu-l depăşesc pe n:

tprim[0]=1, tprim[1]=2, tprim[2]=3, tprim[3]=5 etc.

El este sortat crescător şi ultimul element tprim[m] este cel mai mare număr prim care nu-l depăşeşte pe *n*:

tprim[m] <= n.

Funcţia *prim* returnează numărul numerelor prime păstrate în *tprim* (adică m+1).

Pentru determinarea numerelor prime vom proceda ca mai jos:

Primele 3 numere prime (1, 2 şi 3) se păstrează în mod automat, dacă n>3.

Pentru a determina că un număr m > 2 este prim, este suficient să stabilim că el nu este divizibil cu nici unul din numerele prime mai mici decît el. De fapt, este suficient să testăm divizibilitatea lui cu numerele prime ale căror pătrate nu-l depăşesc. De asemenea, vom observa că singurul număr prim par este 2. Deci un număr m > 2 este prim dacă:

1. *m* este impar.

2. *m* nu se divide cu nici unul din numerele prime consecutive *p1, p2*, ..., *pk*, ale căror pătrate nu-l depăşesc pe *m* (p1=3).

   Deci *pk* este cel mai mare număr prim pentru care pk * pk < m.

Dacă tabloul conţine deja *i* numere prime:

tprim[0], tprim[1], ..., tprim[i-1] (i > 2)

atunci pentru a determina numărul prim următor vom proceda astfel:

1. tprim[i]=tprim[i-1]+2, deoarece numerele prime mai mari decît doi nu sînt pare.

2. Se testează dacă tprim[i] este prim; în acest scop se caută dacă există printre numerele prime:

   tprim[2]=3, tprim[3], ...

   vreunul care să-l dividă pe tprim[i].

   Conform celor spuse mai sus, nu trebuie încercate toate elementele tabloului *tab* ci numai acele elemente care satisfac relaţia:

   (1) tprim[j]*tprim[j] <= tprim[i] (j=2, 3, ...).

   În cazul în care există un element care satisface atît relaţia (1), cît şi relaţia:

   (2) tprim[i]%tprim[j]==0

   numărul tprim[i]nu este prim şi se va trece la pasul 3 de mai jos.

   Altfel tab[i] este prim, se măreşte *i* cu o unitate şi procesul de calcul se reia de la punctul 1.

3. Cum tprim[i] nu este prim, acesta se măreşte cu 2 şi procesul se reia de la punctul 2.

   Procesul de calcul de mai sus se continuă atît timp cît tprim[i] <= n.

După generarea numerelor prime, programul citeşte un întreg şi dacă acesta aparţine intervalului (0,1000), îl caută în tabela de numere prime. Apoi se afişează numărul respectiv însoţit de un mesaj corespunzător. După afişare se revine la citirea numărului următor cu care se procedează analog. Execuţia programului se întrerupe la întîlnirea unui caracter care nu aparţine unui întreg.

Căutarea numărului citit în tabloul de numere prime se face prin metoda căutării binare. În acest scop se utilizează o funcţie analogă cu funcţia *dcautbin* definită în exerciţiul precedent. În acest caz se va folosi funcţia

icautbin. Diferenţa între cele două funcţii constă în aceea că *dcautbin* caută o valoare de tip *double* într-un tablou de tip *double*, iar *icautbin* caută o valoare de tip *int* într-un tablou de tip *int*.

PROGRAMUL BIV43

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 1000

int icautbin(int tab[], int n, int a)
/* cauta in tab un element de valoare egala cu a; returneaza indicele
   elementului respectiv sau -1, daca nu exista un astfel de element */
{
 int i,inf,sup;

 inf = 0;
 sup = n-1;
 while(inf <= sup ) {
     i =(inf+sup)/2;
     if(tab[i] == a)
         return i;
     if(tab[i] > a)
         sup = i-1;
     else
         inf = i+1;
 } /* sfirsit while */

 return -1;
} /* sfirsit icautbin */

int prim( int tprim[], int n)
/* pastreaza in tprim numerele prime care nu-l depasesc pe n */
{
 int i,j,k;

 tprim[0] = 1;
 if( n <= 1 )
    return 1;
 tprim[1] = 2;
 if( n <= 2 )
    return 2;
```

```
tprim[2] = 3;
if( n <= 3)
    return 3;
tprim[3] = 5;
i = 3;

while(tprim[i] <= n) {
/* se cauta existenta unui divizor prim >= 3 pentru tprim[i] */
    j = 2;
    while((k=tprim[j]*tprim[j]) < tprim[i] &&
          tprim[i]%tprim[j])

/* ciclul continua atit timp cit tprim[j]*tprim[j] < tprim[i] si restul
   impartirii lui tprim[i] la tprim[j] este diferit de zero */
        j++;

/* ciclul while se termina cind cel putin una din conditiile de mai sus nu
   este adevarata */
    if( k > tprim[i] ) {

/* tprim[i] este prim deoarece nu exista un divizor prim a lui tprim[i] al
   carui patrat sa fie mai mic sau egal cu tprim[i] */
        i++;
        tprim[i] = tprim[i-1] + 2;

/* tprim[i-1] fiind prim, urmatorul numar prim va fi cel putin cu 2 mai
   mare decit el */

    } /* sfirsit if pentru tprim[i] prim */

    else /* tprim[i]  nu este prim deoarece s-a determinat un j
            asa incit
                tprim[j]*tprim[j] = tprim[i]
            sau
                restul impartirii lui tprim[i] la tprim[j] este zero;
            in ambele cazuri tprim[i] nu este prim;
            se mareste cu 2 */

        tprim[i] += 2;

 } /* sfirsit while exterior */
 return i;
```

```
} /* sfirsit prim */

main() /* citeste un sir de intregi si-i afiseaza pe cei din intervalul
          (0,1000), care sint numere prime */
{
 int nrprime[MAX];
 int nrcrt;
 int n;

 n=prim(nrprime,MAX);

/* n este numarul numerelor prime care nu-l depasesc pe 1000 */
 do {
     printf("Tastati un intreg de cel mult 3\
           cifre:");
     if(scanf("%d", &nrcrt) != 1)
        exit(0);
     if(nrcrt == 1 || nrcrt == 2 || nrcrt == 3 ) {
        printf("%d este un numar prim din\
              intervalul (0,1000)\n", nrcrt);
        continue;
     }
     if(nrcrt>3 && nrcrt<1000 &&
           icautbin(nrprime,n,nrcrt) != -1)
        printf("%d este un numar prim din\
              intervalul (0,1000)\n", nrcrt);
     else
        printf("%d nu este un numar prim din\
               intervalul (0,1000)\n", nrcrt);
  } while(1);
}/* sfirsit main */
```

Observaţie:

Condiţia

tprim[j]*tprim[j]<tprim[i]

poate fi înlocuită cu echivalenta ei:

tprim[j]<sqrt(tprim[i])

care este mai eficientă pentru numere relativ mari, dacă se calculează rădăcina pătrată în afara ciclului *while* interior:

```
...
while (tprim[i] <= n){
```

```
j=2;
k=sqrt(tprim[i]);
while(tprim[j] < k && tprim[i] % tprim[j])
    j++;
    if(tprim[j] > k){
        ...
```

5. CLASE DE MEMORIE

La compilarea unui text sursă, compilatorul alocă memorie pentru variabilele programului. Se pot aloca variabile pe stivă, în regiştrii calculatorului, cît şi în alte zone de memorie.

Programatorul defineşte, cu ajutorul declaraţiilor, modul de alocare al variabilelor. Se pot defini variabile utilizabile (*vizibile*) în *tot* programul, precum şi variabile care au utilizări *locale* (cu vizibilitate *restrînsă*). Variabilele care pot fi utilizate în tot programul se numesc variabile *globale*, iar cele care pot fi utilizate numai într-o anumită parte a programului se numesc variabile *locale*.

5.1. Variabile globale

Variabilele globale au o *definiţie* şi eventual una sau mai multe declaraţii de variabilă *externă*.

Definiţia unei variabile globale coincide cu o declaraţie obişnuită care însă este scrisă în afara corpului oricărei funcţii a programului.

O astfel de definiţie, de obicei, se scrie la începutul unui fişier sursă. Aceasta deoarece ea este valabilă din locul în care este scrisă şi pînă la sfîrşitul fişierului sursă respectiv.

În cazul în care programul se compune din mai multe fişiere sursă, o variabilă globală poate fi utilizată într-un fişier sursă în care nu este definită dacă este declarată ca variabilă externă în acel fişier. O declaraţie de *variabilă externă* coincide cu o declaraţie de variabilă obişnuită precedată de cuvîntul cheie *extern*.

Exemplu:

Un program se compune din două fişiere sursă, *fis1.cpp* şi *fis2.cpp*. În fişierul *fis1.cpp* se definesc variabilele globale *zi*, *luna* şi *an* de tip *int*. Aceste variabile se utilizează atît în *fis1.cpp*, cît şi în *fis2.cpp*. Dacă aceste fişiere se compilează separat, folosind un fişier de tip *project* sau dacă *fis2.cpp* se include în fişierul *fis1.cpp* înainte de a defini variabilele globale respective, atunci ele nu sînt definite în *fis2.cpp* şi în consecinţă nu pot fi utilizate în funcţiile din fişierul *fis2.cpp*. Pentru a elimina acest neajuns, vom declara variabilele respective ca externe în funcţiile din fişierul *fis2.cpp*.

Fişierul fis1.cpp

```
int zi,luna,an; /*Definitie de variabile globale.
                  Aceste variabile pot fi folosite in toate
                  functiile din fisierul fis1.cpp care
                  urmeaza după aceasta definitie */
...
tip f(...)
{
 int i;
 ...
 i=an % 4==0 && an % 100 != 0 || an % 400==0;
 ...
 if(luna==2)
     zi=28+i;
 ...
}

tip g(...)
{
 ...
 if(zi <= 27)
     zi++;
 ...
 if(zi==31&&luna==12)
     an++;
 ...
}
```

Fişierul fis2.cpp

```
...

main()
{
 extern int zi,luna,an; /* declaratie de extern pentru
                           variabilele globale zi, luna si an,
                           definite in fis1.cpp;
                           este valabila in corpul functiei
                           main */
 ...
 if (zi<1||zi>31)
```

```
        printf("ziua eronata\n");
    if(luna<1||luna>12)
        printf("luna eronata\n");
    if(an<1600||an>4900)
        printf("anul eronat\n");
  ...
  } /* sfirsit main */

  tip p(...)
  {
    ... /* nu exista declaratii de extern pentru variabilele
           globale zi,luna,an; in acest caz nu se pot utiliza variabilele
             respective numai daca fis2.cpp se include in fis1.cpp
            dupa definitia acestor variabile globale */
    ...
    zi=15;  /* zi apare la compilare ca nedefinita */
    ...
  }
```

Observaţie:

Întrucît repartizarea funcţiilor în fişiere poate să se modifice pe măsură ce se construieşte programul, se recomandă ca orice utilizare a unei variabile globale să fie precedată de declaraţia de *extern* a ei.

Variabilele globale sînt alocate în momentul compilării. Lor li se alocă memorie, la întîlnirea definiţiilor lor, într-o zonă de memorie prevăzută special pentru variabilele globale ale programului.

5.2. Variabile locale

Variabilele locale, spre deosebire de cele globale, nu sînt valabile în tot programul. Ele au o valabilitate locală, în unitatea în care sînt declarate.

Variabilele locale pot fi alocate pe *stivă*. În acest caz ele se numesc *automatice*. Acestea se declară în mod obişnuit, în corpul unei funcţii sau la începutul unei instrucţiuni compuse. O astfel de variabilă se alocă la execuţie (nu la compilare).

La apelul unei funcţii, variabilele automatice (declarate în mod obişnuit înaintea primei instrucţiuni din corpul funcţiei respective) se alocă pe stivă. În momentul în care se revine din funcţie, variabilele automatice alocate la apel, se dezalocă (elimină) şi stiva revine la starea dinaintea apelului (operaţia de curăţire a stivei). Aceasta înseamnă că variabilele automatice îşi pierd existenţa la revenirea din funcţia în care sînt declarate. De aceea, o

variabilă automatică este valabilă (vizibilă) numai în corpul funcţiei în care a fost declarată.

În acelaşi mod se comportă variabilele automatice declarate la începutul unei instrucţiuni compuse. O astfel de variabilă se alocă pe stivă în momentul în care controlul programului ajunge la instrucţiunea compusă în care este declarată variabila respectivă şi se elimină de pe stivă în momentul în care controlul programului trece la instrucţiunea următoare celei compuse.

Variabilele locale pot şi să nu fie alocate pe stivă. În acest scop ele se declară ca fiind *statice*. O declaraţie de variabilă statică este o declaraţie obişnuită precedată de cuvîntul cheie *static*.

Variabilele statice pot fi declarate atît în corpul unei funcţii cît şi în afara corpului oricărei funcţii.

O variabilă statică declarată în corpul unei funcţii este definită numai în corpul funcţiei respective.

Spre deosebire de variabilele automatice, o variabilă statică nu se alocă pe stivă la execuţie, ci la compilare într-o zonă de memorie destinată acestora.

O variabilă statică declarată în afara corpurilor funcţiilor este definită (vizibilă) din punctul în care este declarată şi pînă la sfîrşitul fişierului sursă care conţine declaraţia respectivă. Spre deosebire de variabilele globale, o astfel de variabilă nu poate fi declarată ca externă. Deci ea nu poate fi utilizată în alte fişiere dacă acestea se compilează separat sau se includ înaintea declaraţiei respective.

Putem spune că o variabilă statică declarată în afara corpurilor funcţiilor este locală fişierului sursă în care este declarată.

Ea se alocă la compilare într-o zonă specială rezervată variabilelor statice corespunzătoare fişierului sursă în care au fost declarate.

Dacă prin *modul* înţelegem un text sursă compilat separat faţă de restul textului sursă al unui program, atunci putem afirma că o variabilă statică declarată în afara corpurilor funcţiilor este locală modulului în care a fost declarată. La aceasta mai trebuie să adăugăm şi faptul că utilizările unei variabile statice trebuie să fie precedate de declaraţia ei.

În cazul variabilelor globale, acestea pot fi utilizate în orice modul, dacă sînt precedate, în modulul respectiv, fie de definiţia lor, fie de declaraţia de extern.

Funcţiile sînt analoge cu variabilele globale. Ele, ca şi variabilele globale, pot fi utilizate (apelate) în orice modul al programului dacă apelul lor este

precedat, în modulul respectiv, fie de definiţie, fie de prototipul lor. În acest caz, prototipul funcţiei înlocuieşte declaraţia de *extern* a variabilei globale.

Limbajele C şi C++ permit declarări de funcţii statice. Aceasta se face precedînd antetul funcţiei de cuvîntul cheie *static*. În acest caz, funcţia respectivă este locală modulului în care este definită.

Împărţirea pe module a unui program nu se face prin reguli definite riguros.

Se obişnuieşte să se spună că un modul este format din funcţii "înrudite" şi conţine date pe care acestea le prelucrează în comun.

Exemplu:

Un program se compune din 2 fişiere *sursa1.cpp* şi *sursa2.cpp* care formează fiecare cîte un modul (se compilează separat, utilizînd, de exemplu, un fişier de tip *project*).

Fişierul sursa1.cpp conţine:

```
int i,j,k; /* - definitie de variabile globale;
              - se pot utiliza in tot modulul;
              - se pot declara ca externe in celalalt modul pentru
                a putea fi utlizate si acolo */

static double x,y; /* - declaratie de variabile statice;
                      - sint utilizabile in tot modulul de fata
                        si numai in acesta */
...

main()
{
 int p,q; /*- declaratie de variabile automatice;
             - sint utilizabile numai in corpul functiei main */

 static int u,v,r; /*- declaratie de variabile statice;
                      - sint utilizabile numai in corpul functiei
                        main */
 ...
/* se pot utiliza variabilele i, j, k, x, y, p, q, u, v si r */
 ...
}
```

```
tip f(...)
{
 float a,b;  /*- declaratie de variabile automatice;
                 - sint utilizabile numai in corpul functiei f */

 static char c,temp; /*- declaratie de variabile statice;
                          - sint utilizabile numai in corpul
                            functiei f */
 ...

/* se pot utiliza variabilele i, j, k, x, y, a, b, c si temp */
}
```

Fişierul sursa2.cpp conţine:

```
static float s,t; /*- declaratie de variabile statice;
                      - sint utilizabile in fisierul sursa2.cpp */
tip g(...)
{
 extern int i,j,k; /* - declaratie de extern pentru variabilele
                         i, j, k;
                       - sint utilizabile in corpul  functiei g */

 char d,e; /* - declaratie de variabile automatice;
               - sînt utilizabile in corpul functiei g */

 static double d1,d2,d3; /*- declaratie de variabile statice;
                             - sint utilizabile in corpul
                               functiei g */

 ...

/* se pot utiliza variabilele i, j, k, s, t, d, e, d1, d2 si d3 */
 ...
}

tip h(...)
{
 extern int k; /* - declaratie de extern pentru variabila k;
                   - este utilizabila in corpul functiei h */

 ...
```

```
/* se pot utiliza variabilele k, s si t */
  ...
}
```

Variabilele pot fi *redeclarate* indiferent de modul lor de alocare (global, static sau automatic).

Exemplu:

```
int i,c; /* variabile globale */
static float a; /* variabila statica */
...
main()
{
 double c; /* variabila automatica diferita de variabila globala c */
 char a; /* variabila automatica diferita de variabila statica a */
 ...
 c=123;     /* se atribuie variabilei automatice c valoarea 123 dupa
               conversia ei in flotanta dubla precizie */
 ...
 i=123; /* se atribuie variabilei globale i valoarea 123 */
 ...
 a='1'; /* se atribuie variabilei automatice a codul ASCII
           al caracterului 1 */

/* - in aceasta functie nu se pot utiliza variabila globala c si variabila
     statica a deoarece acestea au fost redeclarate;
   - cu toate acestea, in limbajul C++ este posibila utilizarea lor
     folosind operatorul :: pentru globale. */
}

tip f(...)
{
...
/* nu sînt redeclarari pentru i,c si a */

 c=123; /* se atibuie variabilei globale c valoarea 123 */
 a=123; /* se atribuie variabilei statice a valoarea 123, dupa
           ce a fost convertita spre flotant simpla precizie */
 ...
}
```

Uneori, tablourile de dimensiuni mari se alocă globale sau statice

deoarece alocarea lor pe stivă poate conduce la depăşirea stivei.

5.3. Alocarea parametrilor

Parametrii formali se alocă pe stivă ca şi variabile automatice. De aceea, ei se consideră a fi variabile locale şi sînt utilizabili numai în corpul funcţiei în antetul căreia sînt declaraţi.

La apelul unei funcţii, se alocă pe stivă parametri formali, dacă există, li se atribuie valorile parametrilor efectivi care le corespund. Apoi se alocă pe stivă variabilele automatice declarate la începutul corpului funcţiei respective.

La revenirea din funcţie, se realizează curăţirea stivei, adică sînt eliminate de pe stivă (dezalocate) atît variabilele automatice, cît şi parametri. În felul acesta, la revenirea din funcţie, stiva ajunge la starea dinaintea apelului.

5.4. Utilizarea parametrilor şi a variabilelor globale

Am văzut că parametrii efectivi, se utilizează pentru a concretiza valorile care intervin în procesul de calcul definit printr-o funcţie. Se obişnuieşte să se spună că valorile respective se "transferă" funcţiei prin intermediul parametrilor.

Din această cauză, parametri se mai consideră ca fiind o "interfaţă" între funcţii. În limbajul C această interfaţă este unilaterală, adică ea transferă date de la funcţia care face apelul, la funcţia apelată.

Variabilele globale reprezintă şi ele o interfaţă între funcţii. În acest caz nu se transferă date de la o funcţie la alta, ci orice funcţie are acces la valorile variabilelor globale. Dacă o funcţie trebuie să transfere o valoare funcţiei apelate, atunci aceasta are două posibilităţi:

1. Să folosească un parametru la care i se atribuie valoarea respectivă înainte de apel.
2. Să atribuie valoarea respectivă unei variabile globale înainte de apel.

În primul caz, funcţia apelată are destinat un parametru formal la care i se atribuie în mod automat valoarea respectivă la apel. În al doilea caz, funcţia apelată trebuie să aibă acces la variabila globală căreia i s-a atribuit valoarea de transferat. În acest scop, este necesar să se cunoască *numele* variabilei globale respective în momentul în care se programează funcţia

respectivă. Aceasta este un neajuns al variabilelor globale faţă de parametri. Numele şi sensul variabilelor globale trebuie să fie cunoscut înainte de a începe programarea diferitelor funcţii care utilizează în comun variabilele respective.

Un alt dezavantaj al variabilelor globale este sursa mare de erori pe care o reprezintă. Într-adevăr, orice funcţie din program avînd acces la o variabilă globală, ea poate modifica valoarea acesteia şi de aceea posibilitatea modificării eronate este mare. De asemenea, depistarea unei astfel de erori este destul de complicată, deoarece este necesar să se studieze un număr mare de funcţii (toate funcţiile care au acces la variabila globală a cărei valoare s-a constat că este eronată).

În cazul utilizării parametrilor, se realizează o protecţie a datelor, deoarece funcţia apelată nu poate modifica valorile parametrilor efectivi în mod direct.

Avantajul variabilelor globale decurge din faptul că ele reprezintă o interfaţă simplă între funcţii, interfaţă care este în ambele sensuri. O funcţie poate modifica valoarea unei variabile globale, modificare care rămîne valabilă la revenirea din ea, deci funcţia care a făcut apelul poate beneficia de modificarea făcută prin intermediul funcţiei apelate. Dar chiar acest fapt se consideră că este o sursă de erori. De aceea, se consideră că nu este bine să se exagereze cu utilizarea variabilelor globale, ele constituind o sursă de erori.

De obicei, folosim variabilele globale cînd rezultatele unei funcţii sînt folosite în comun de mai multe funcţii ale programului. În rest, se recomandă utilizarea parametrilor pentru realizarea interfeţelor dintre funcţii.

5.5. Variabile registru

O variabilă alocată într-un registru al calculatorului se numeşte *variabilă registru*. Ea se declară printr-o declaraţie obişnuită precedată de cuvîntul cheie *register*.

Se pot declara, ca variabile registru, numai variabilele de tip *int*, *char* şi *pointer* (acest tip va fi introdus într-un capitol următor). De asemenea, numai parametrii şi variabilele automatice simple pot fi declarate ca variabile registru.

Numai un număr limitat de variable se pot aloca în regiştri. Alocarea se face la apelul unei funcţii şi în ordinea în care au fost declarate aceste variabile în funcţia respectivă. Alocarea rămîne valabilă pînă la revenirea din funcţie. În cazul în care nu pot fi alocate în regiştri toate variabilele

registru declarate de programator, se alocă cîte se pot în ordinea declarării lor, iar restul se alocă în mod obişnuit pe stivă, ca orice variabilă automatică sau parametru.

Se recomandă să se declare ca variabile registru, acele variabile ale unei funcţii care au o utilizare mare în funcţia respectivă. Alocarea unei variabile în registru conduce la economie de memorie, precum şi la creşterea vitezei de calcul.

Menţionăm că, în lipsa declaraţiilor registru, compilatorul alocă în mod implicit anumite variabile automatice sau parametri în regiştri.

În legătură cu variabilele registru amintim că lor nu li se poate aplica operatorul adresă (&).

Exerciţii:

5.1 Să se scrie o funcţie care citeşte:

- valoarea variabilei *m* de tip int;
- valoarea variabilei *n* de tip int;
- m*n numere care reprezintă elementele unei matrice de ordinul m*n.

Funcţia are următorii parametrii:

dmat - Tablou care păstrează elementele matricei citite.

max - Maximul produsului m*n admis.

Funcţia returnează produsul m*n şi atribuie valorile lui *m* şi *n*, variabilelor globale *nrlin*, respectiv *nrcol*.

Tabloul *dmat* este unidimensional, matricea păstrîndu-se prin *liniarizare*.

Dacă notăm cu a[i,j], i=0,1,...m-1; j=0,1,...,n-1; elementele matricei, atunci elementul a[i,j] se atribuie lui dmat[k], unde:

(1) k=i*n+j

Această relaţie poartă denumirea de relaţie de *liniarizare*. Ea permite păstrarea pe linii a elementelor matricei citite. Astfel, prima linie a matricei este formată din elementele a[0,j], pentru j=0,1,...,n-1. Conform relaţiei (1), aceste elemente se păstrează în tabloul *dmat* prin intermediul elementelor:

dmat[0], dmat[1], ..., dmat[n-1].

Linia a doua, adică elementele a[1,j], pentru j=0,1,...,n-1, se păstrează în tabloul *dmat* ca elemente ce au indicii k=1.n+j, adică:

dmat[n], dmat[n+1], ..., dmat[n+n-1].

În general, linia formată din elementele a[i,j] pentru j=0,1,...,n-1 se păstrează în tabloul *dmat* ca:

dmat[i*n], dmat[i*n+1], ..., dmat[i*n+n-1]

Funcţia de faţă foloseşte funcţia *ndcit* definită în exerciţiul 4.37.

FUNCŢIA BV1

```
int gdcitmat(double dmat[], int max)
/* citeste
   m-numar de linii;
    n-numar de coloane;
    m*n numere de tip double pe
  care le pastreaza in matricea dmat prin liniarizare;
  returneaza valoarea m*n;
  realizeaza atribuirile:
    nrlin=m;
    nrcol=n; */
{
 extern int nrlin,nrcol;
 int i,m,n;
 char t[255];

 do { /* se citesc valorile lui m si n */
     do { /* citeste pe m */
         printf("numarul de linii= ");
         if(gets(t) == NULL ) {
             printf("s-a tastat EOF\n");
             exit(1);
         }
         if(sscanf(t,"%d", &m) == 1 && m > 0 &&
                 m <= max)
             break;
         printf("nu s-a tastat un intreg in\
                intervalul [1,%d]\n",max);
     } while(1);
     do { /* citeste pe n */
         printf("numarul de coloane= ");
         if(gets(t) == NULL ) {
             printf("s-a tastat EOF\n");
             exit(1);
```

```
          }
          if(sscanf(t,"%d",&n) == 1 && n > 0 &&
           n <= max )
             break;
          printf("nu s-a tastat un intreg in\
                intervalul [1,%d]\n",max);
      } while (1);
      i = m*n;
      if(i <=max)
         break;
      printf("produsul m*n=%d depaseste pe\
           max=%d\n", i, max );
      printf("se reiau citirile lui m si n\n");
   } while(1);

  /* se citesc elementele matricei tastate pe linii */
   if(ndcit(i,dmat) != i ) {
      printf("nu s-au tastat %d elemente\n", i);
      exit(1);
   }
   nrlin = m;
   nrcol = n;
   return i;
  }
```

5.2 Să se scrie o funcţie care calculează produsul dintre o matrice de ordinul m*n şi o matrice coloană (de ordinul n*1).

Funcţia are următorii parametrii:

m	- Numărul liniilor matricei.
n	- Numărul coloanelor matricei.
dmat	- Tablou unidimensional de tip *double* care păstrează elementele matricei de ordinul m*n prin liniarizare.
dmatcol	- Tablou unidimensional de tip *double* care păstrează elementele matricei coloana.
dmatrez	- Tablou unidimensional de tip *double* care păstrează elementele matricei rezultat.

Matricea rezultat este de ordinul m*1. Elementele ei se calculează prin relaţia de mai jos:

dmatrez[i]=a[i,0]*dmatcol[0]+a[i,1]*dmatcol[1]+...+
a[i,n-1]*dmatcol[n-1]

pentru i=0,1,...,m-1, unde prin a[i,j] am notat un element al matricei de ordinul m*n. Ţinînd seama de relaţia de liniarizare, suma de mai sus se scrie astfel:

dmatrez[i]=dmat[i*n]*dmatcol[0]+
dmat[i*n+1]*dmatcol[1]+...+ dmat[i*n+n-1]*dmatcol[n-1].

FUNCŢIA BV2

```
void pmatcol(int m, int n, double dmat[],
             double dmatcol[],double dmatrez[])

/* calculeaza produsul dintre matricea dmat de ordinul m*n si
   vectorul coloana dmatcol de ordinul n*1; se obtine matricea
   coloana dmatcol de ordinul m*1 */
{
 int i,j,k;

 for(i=0; i < m; i++) {
     dmatrez[i] =0;
     k=i*n;
     for(j=0; j <= n-1; j++)
         dmatrez[i] += dmat[k+j] * dmatcol[j];
 }
}
```

5.3 Să se scrie un program care citeşte elementele unei matrice *a* de ordinul m*n, elementele unei matrice coloane *b* de ordinul n*1, calculează şi afişează produsul a*b.

În acest scop se vor apela trei funcţii:

- funcţia *gdcitmat* care citeşte elementele matricei *a* (vezi exerciţiul 5.1.);
- funcţia *dvcit* care citeşte elementele matricei *b* (vezi exerciţiul 4.38.);
- funcţia *pmatcol* care calculează produsul a*b (vezi exerciţiul 5.2.).

Deoarece funcţiile *gdcitmat* şi *dvcit* apelează funcţia *ndcit*, se va include şi fişierul care conţine definiţia acestei funcţii (vezi exerciţiul 4.37.).

Matricea *a* va fi precedată de doi întregi care reprezintă valorile lui *m* şi *n*.

Matricea *b* va fi precedată de un întreg care reprezintă valoarea lui *n*. Această valoare, deşi pare că nu este necesară înaintea elementelor matricei *b*, ea se tastează pe de o parte pentru a face o verificare cu privire la numărul

datelor tastate, iar pe de altă parte, prezenţa ei este cerută de funcţia *dvcit*.

PROGRAMUL BV3

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include "biv37.cpp" /* functia ndcit */
#include "biv38.cpp" /* functia dvcit */
#include "bv1.cpp" /* functia gdcitmat */
#include "bv2.cpp" /* functia pmatcol */

int nrlin,nrcol; /* variabile globale utilizate la citirea matricei
                    cu ajutorul functiei gdcitmat */

#define MAX 1000

main() /* citeste elementele matricei a de ordinul m*n si elementele
          matricei b de ordinul n*1, calculeaza si afiseaza produsul
          c = a*b */
{
 int m;
 double a[MAX],b[MAX],c[MAX];

 gdcitmat(a,MAX); /* citeste matricea a; la revenire, nrlin are
                     ca valoare numarul liniilor (m), iar nrcol
                     numarul coloanelor(n) */
/* citeste matricea b */
     if(dvcit(MAX,b) != nrcol) {
         printf("matricea coloana nu are\
             %d linii\n",nrcol);
         exit(1);
     }

/* calculeaza produsul c = a*b*/
     pmatcol(nrlin,nrcol,a,b,c);

/* listeaza matricea rezultat*/
     for( m = 0; m < nrlin; m++) {
         printf("c[%d]=%g\n",m,c[m]);
         if((m+1)%23 == 0 ) {
             printf("actionati o tasta pentru a\
                    continua\n");
```

```
                getch();
            }
        }
    }
```

5.4 Să se scrie o funcţie care calculează produsul a două matrice cu elemente de tip *double*.

Fie *a* o matrice de ordinul m*n şi *b* o matrice de ordinul n*s. Matricea:

c=a*b

este de ordinul m*s. Elementele ei se calculează folosind relaţia:

c[i,j]=a[i,0]*b[0,j]+a[i,1]*b[1,j]+...+a[i,n-1]*b[n-1,j]

pentru i=0,1,2,...,m-1 şi j=0,1,2,...,s-1.

Elementele celor trei matrice se păstrează liniarizat în tablourile unidimensionale *dmata*, *dmatb* şi *dmatc*. Folosind relaţia de liniarizare, suma de mai sus se reprezintă astfel:

```
dmatc[i*s+j]=dmata[i*n]*dmatb[j]+
             dmata[i*n+1]*dmatb[s+j]+...+
             dmata[i*n+k]*dmatb[k*s+j]+...+
             dmata[i*n+n-1]*dmatb[(n-1)*s+j]
```

FUNCŢIA BV4

```
void dprodmat(int m,int n,int s, double dmata[],
              double dmatb[], double dmatc[])
/* calculeaza produsul matricelor a si b:
      c = a*b;
   dmata pastreaza matricea a de ordinul m*n;
   dmatb pastreaza matricea b de ordinul n*s;
   dmatc pastreaza matricea c de ordinul m*s */
{
 int i,j,k;
 int in,is;

 for(i=0; i < m ; i++) {
     is=i*s;
     in= i*n;
     for(j=0; j<s; j++) {
         dmatc[is+j] = 0;
         for(k=0;k<n;k++)
             dmatc[is+j] += dmata[in+k]*dmatb[k*s+j];
```

```
        }
    }
}
```

5.5 Să se scrie un program care citeşte elementele a două matrice de tip *double*, calculează şi afişează produsul lor.

Programul utilizează funcţia *gdcitmat* pentru a citi elementele celor două matrice. De aceea, elementele matricelor vor fi precedate de ordinul lor (numărul de linii şi numărul coloanelor - vezi exerciţiul 5.1.).

PROGRAMUL BV5

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include "biv37.cpp" /* functia ndcit */
#include "bv1.cpp" /* functia gdcitmat */
#include "bv4.cpp" /* functia dprodmat */

#define MAX 1000

int nrlin,nrcol;

main() /* citeste elementele a doua matrice, calculeaza si
          afiseaza matricea produs */
{
 int m,n,s;
 double a[MAX],b[MAX],c[MAX];
 int i,j,is,k;

/* citeste matricea a*/
 gdcitmat(a,MAX);
 m=nrlin; /* numarul liniilor matricei a */
 n=nrcol; /* numarul coloanelor matricei a */

/* citeste matricea b */
 gdcitmat(b,MAX);
 if(nrlin != n) {
    printf("matricea a are %d coloane\n",n);
    printf("matricea b are %d linii\n",nrlin);
    exit(1);
 }
```

```
 s=nrcol;  /* nr coloanelor matricei b */

/* se realizeaza produsul c = a*b */
 dprodmat(m,n,s,a,b,c);

/* afiseaza elementele matricei produs*/
 k=0;
 for(i=0;i<m;i++) {
     printf("\n\tlinia nr.%d\n",i+1);
     is=i*s;
     k++;
     for(j=0;j<s;j++) {
         printf("%g ", c[is+j]);
         if(j%5 == 4) {

/* afiseaza 5 elemente pe un rind */

               printf("\n");
               k++; /* numara rindurile */
         }
         if(k==23) {
             printf("actionati o tasta pentru a\
                     continua\n");
             getch();
             k=1;
         }
     }
   }
}
```

6. INIŢIALIZARE

Adesea dorim ca unele variabile din program să aibă valori iniţiale. Acest lucru este posibil şi anume, variabilelor globale li se pot da valori iniţiale la definirea lor, iar celelalte clase de variabile pot fi iniţializate la declararea lor.

6.1. Iniţializarea variabilelor simple

O variabilă simplă poate fi iniţializată printr-o construcţie de forma:

tip nume=expresie;

sau

static *tip nume=expresie;*

dacă variabila este statică.

Prima formă corespunde variabilelor automatice sau globale.

Variabilelor globale sau statice li se atribuie valori iniţiale la compilare, deci ele au valorile respective la lansarea programului. Faptul că aceste variabile se iniţializează la compilare, cere ca expresiile utilizate pentru iniţializare să fie constante.

Variabilele automatice simple se iniţializează la execuţie, de fiecare dată cînd ele se alocă pe stivă, adică la apelul funcţiei care conţine declaraţiile lor. În acest caz,expresiile utilizate pentru iniţializare pot să nu fie constante. Cu toate acestea, operanzii variabili ai unei astfel de expresii trebuie să aibă valori definite în prealabil. În caz contrar, expresia de iniţializare nu poate fi evaluată în momentul în care se alocă pe stivă variabila automatică pe care o iniţializează.

Variabilele globale şi statice neiniţializate au în mod implicit valoarea *zero*. Acest lucru nu mai are loc în cazul variabilelor automatice.

O variabilă automatică neiniţializată are o valoare imprevizibilă în momentul apelului funcţiei în al cărui corp este declarată. Aceasta rezultă din faptul că ea se alocă pe stivă şi în lipsa expresiei de iniţializare în declaraţia sa, ei nu i se atribuie nici o valoare. În felul acesta, valoarea ei rămîne nedefinită pînă în momentul în care i se atribuie o valoare printr-o instrucţiune de atribuire.

În toate cazurile se fac conversii dacă tipul expresiei de iniţializare nu coincide cu tipul variabilei pe care o iniţializează.

Exemple:

1.

```
int n=100; /*- definitie de variabila globala;
              - lui n i se atribuie valoarea 100;
              - la lansarea programului n are valoarea 100. */
```

2.

```
double x=3; /*- definitie de variabila globala;
              - lui x i se atribuie valoarea 3 dupa ce se converteste
                spre double;
              - la lansarea programului x are valoarea 3. */
```

3.

```
#define MAX 100

int i = MAX*2; /* - i este variabila globala initializata cu
                    valoarea 200;
                  - la lansarea programului i are valoarea 200. */
```

4.

```
int k; /*- definitie de variabila globala;
         - la lansarea programului k are valoarea zero. */
```

5.

```
static char c='a'; /*- declaratie de variabila statica;
                     - lui c i se atribuie codul ASCII al ca-
                       racterului a(97);
                     - la lansarea programului c are valoarea 97. */
```

6.

```
static int s; /*- declaratie de variabila statica;
                - la lansarea programului s are valoarea zero. */
```

7.

```
static int p='a'+1; /*- declaratie de variabila statica;
                      - la lansarea programului s are ca valoare
                        codul ASCII al literei b (97+1). */
```

8.

```
tip f(int n)
{
 int x=123; /* - declaratie de variabila automatica;
              - la fiecare apel al functiei f, x se aloca
                pe stiva si i se atribuie valoarea 123. */
 int a=x+n; /*- declaratie de variabila automatica;
             - la fiecare apel al functiei f, a se aloca
               pe stiva si i se atribuie valoarea expresiei x+n;
             - la intilnirea declaratiei ambii operanzi ai
               expresiei x+n sint definiti. */
```

Exerciţii:

6.1 Să se scrie o funcţie care are ca parametri doi întregi *x* şi *y*, calculează şi returnează numărul aranjamentelor de *x* obiecte luate cîte *y*.

Funcţia de faţă este analogă funcţiei definite în exerciţiul 4.29.

FUNCŢIA BVI1

```
double aranjamente ( int x, int y)
/* calculeaza si returneaza numarul aranjamentelor de
   x obiecte luate cite y */
{
 double a=1.0;
 int i=x-y+1;

 if(x < 1 || x > 170 )
    return -1.0;
 if(y < 1 || y > x )
    return -1.0;
 while( i <= x )
    a *= i++;
 return a;
}
```

Observaţie:

Pentru a testa funcţia de faţă se poate utiliza programul din exerciţiul 4.30. Programul respectiv apelează funcţia *aranjamente* definită în exerciţiul 4.29.

Pentru a apela funcţia *aranjamente* definită mai sus, în programul din exerciţiul 4.30. se va schimba construcţia

#include "BIV29.CPP"

cu

#include "BVI1.CPP"

6.2. Inițializarea tablourilor

Tablourile, ca și variabilele simple, pot fi inițializate. Tablourile *globale* se inițializează prin *definițiile* lor. Tablourile *statice* și *automatice* se inițializează prin *declarațiile* lor.

În limbajul C, în toate cazurile, inițializările se fac prin expresii *constante*.

Tablourile *globale* și *statice* se inițializează la *compilare*. De aceea, la lansarea programului, elementele lor au ca valori, valorile expresiilor cu care au fost inițializate.

Tablourile *automatice* se inițializează la *execuție*, de fiecare dată cînd se apelează funcția în care sînt declarate. Menționăm că în unele versiuni ale limbajului C nu se pot inițializa tablourile automatice.

Limbajul Turbo C permite inițializarea tablourilor automatice.

Inițializarea unui tablou unidimensional se realizează printr-o construcție de forma:

tip nume[ec] = {ec0,ec1,...,eci};

sau

static *tip nume [ec] = {ec0,ec1,...,eci};*

dacă tabloul este static.

Prin ec, ec0, ec1, ..., eci am notat expresii constante.

Prin aceste construcții, elementul nume[k] se inițializează cu valoarea expresiei constante eck, pentru k=0,1,2,...i.

În general, i <= ec-1. Dacă i >= ec, atunci construcția este eronată. Dacă i < ec-1, atunci elementele

nume[i+1], nume[i+2], ..., nume[ec-1]

rămîn neinițializate.

Elementele *neinițializate* ale unui tablou *global sau static* au în mod implicit valoarea inițială egală cu *zero*.

Elementele *neiniţializate* ale unui tablou *automatic* au valori iniţiale *nedefinite*.

În cazul în care se iniţializează toate elementele unui tablou unidimensional, se poate omite expresia din parantezele pătrate care defineşte numărul elementelor tabloului. Deci construcţiile:

tip nume[] = {ec0,ec1,...,ecn};

şi

static *tip nume[] = {ec0,ec1,...,ecn};*

sînt corecte şi în ambele cazuri tablourile au n+1 elemente, iar elementul

nume[i]

este iniţializat cu valoarea expresiei *eci*.

Menţionăm că, la fel ca şi în cazul variabilelor simple, dacă expresia de iniţializare are un tip diferit de cel al tabloului, atunci valoarea ei se converteşte spre tipul tabloului înainte de a fi atribuită elementului pe care-l iniţializează.

Observaţie:

Într-o declaraţie sau definiţie de tablou unidimensional se poate omite expresia care defineşte numărul elementelor tabloului în următoarele cazuri:

1. Definiţia sau declaraţia de tablou conţine expresii constante pentru iniţializarea fiecărui element al tabloului.
2. Declaraţia se referă la un tablou unidimensional care este parametru formal.
3. Declaraţia de tablou extern unidimensional:

```
extern int tab[];
```

Exemple:

1. `int tab[10] = {0,1,2,3,4,5,6,7,8,9};`

   Prin această definiţie/declaraţie, lui tab[i] i se atribuie valoarea *i*.

2. `int tab[] = {0,1,2,3,4,5,6,7,8,9};`

   Această construcţie este identică, ca efect,cu cea precedentă.

3. `double d[20] = {0,1,-1,3,-2};`

   Primele 5 elemente ale tabloului *d* se iniţializează astfel:

d[0]=0;
d[1]=1;
d[2]=-1;
d[3]=3;
d[4]=-2.

Celelalte elemente ale tabloului d[5], d[6], ..., d[19] au valoarea inițială zero dacă *d* este un tablou global și o valoare inițială imprevizibilă dacă tabloul este automatic.

4.

```
#define V ('A'-10)
static int chexa[]= {'A'-V,'B'-V,'C'-V,
                     'D'-V,'E'-V,'F'-V};
```

Elementele tabloului *chexa* se inițializează astfel:

chexa[0]='A'-('A'-10)=10
chexa[1]='B'-('A'-10)=11
chexa[2]=12
chexa[3]=13
chexa[4]=14
chexa[5]=15.

5. `char er[] = {'e','r','o','a','r','e','\0'};`

Tabloul are 7 elemente care se inițializează astfel:

er[0]='e'
er[1]='r'
er[2]='o'
er[3]='a'
er[4]='r'
er[5]='e'
er[6]='\0'

Se observă că tabloul *er* păstrează șirul de caractere "eroare", inclusiv caracterul *NUL* care termină orice șir de caractere.

Avînd în vedere faptul că inițializarea tablourilor de tip *char* se utilizează frecvent, autorii limbajului C au introdus o simplificare pentru inițializarea tablourilor de acest tip. Astfel, pentru un tablou de tip *char* se pot utiliza una din construcțiile:

char *nume[ec]=șir;*

sau

static char *nume[ec]=şir;*

unde prin *şir* se înţelege o succesiune de caractere delimitată prin ghilimele. Prin aceste construcţii, elementele tabloului *nume* se iniţializează cu codurile ASCII ale caracterelor din compunerea şirului de caractere *şir*, iar după ultimul caracter al şirului se memorează caracterul *NUL*. Expresia constantă *ec*, dacă este prezentă, atunci ea este cel puţin egală cu numărul caracterelor proprii ale şirului mărit cu unu.

Din cele de mai sus rezultă că iniţializarea tabloului *er* se poate realiza folosind construcţia:

```
char er[]="eroare";
```

Tablourile multidimensionale pot fi şi ele iniţializate prin construcţii analoge. Aşa de exemplu, un tablou bidimensional se poate iniţializa printr-o construcţie de forma:

tip nume [n][m]={
{ec11,ec12,...,ec1m},
{ec21,ec22,...,ec2m},
...
{ecn1,ecn2,...,ecnm}
};

unde:

n, m, ecij - Pentru i=1,2,...,n şi j=1,2,...,m sînt expresii constante.

Numărul expresiilor constante poate fi mai mic decît *m* în oricare din acoladele corespunzătoare celor *n* linii ale tabloului bidimensional.

În cazul tablourilor statice, declaraţia este precedată de cuvîntul cheie *static*.

Într-o declaraţie sau definiţie de tablou bidimensional se poate omite numai expresia constantă din prima paranteză pătrată, adică se poate omite numai *n* din construcţia de mai sus.

În general, într-o declaraţie sau definiţie de tablou multidimensional se poate omite numai expresia constantă din prima paranteză pătrată, adică limita superioară pentru primul indice. Ea poate fi omisă în una din cele trei cazuri care au fost amintite la tablourile unidimensionale:

- la declaraţii sau definiţii care conţin iniţializări;
- la declaraţia de parametri;
- la declaraţii de *extern*.

Exemple:

1.

```
int tab[3][4]={
 {-1,0,1,-1},
 {-1,0,1,2},
 {10,20,30,40}
};
```

Prin această construcţie, elementele tabloului *tab* se iniţializează cu valorile:

tab[0][0]=-1, tab[0][1]=0, tab[0][2]=1, tab[0][3]=-1;
tab[1][0]=-1, tab[1][1]=0, tab[1][2]=1, tab[1][3]=2;
tab[2][0]=10, tab[2][1]=20, tab[2][2]=30, tab[2][3]=40.

2.

```
int tab[][4]={
 {-1,0,1,-1},
 {-1,0,1,2},
 {10,20,30,40}
};
```

Această construcţie este identică, ca efect, cu cea din exemplul precedent.

3.

```
double t[][3]={
 {0,1},
 {-1},
 {1,2,3}
};
```

t reprezintă o matrice de ordinul 3*3 ale cărei elemente se iniţializează astfel:

t[0][0]=0, t[0][1]=1;
t[1][0]=-1;
t[2][0]=1, t[2][1]=2, t[2][2]=3.

Elementele t[0][2], t[1][1] şi t[1][2] nu sînt iniţializate. Ele au valoarea iniţială zero dacă tabloul este global sau valori iniţiale nedefinite dacă tabloul este automatic.

Construcţiile utilizate pentru iniţializarea tablourilor bidimensionale se extind imediat pentru tablouri cu un număr mai mare de dimensiuni.

Exemplu:

```
int t3[][3][2]={
 {{100,200},{-1,1},{0,1}},
 {{123,456},{789,10},{11,12}}
};
```

Exerciţii:

6.2 Să se scrie un program care citeşte un întreg pozitiv ce reprezintă o sumă exprimată în lei. Se cere să se afişeze numărul minim de bancnote şi monede de 10000 lei, 5000 lei, 1000 lei,500 lei, 200 lei etc. necesare pentru a exprima suma respectivă.

Procesul de calcul a fost indicat în exerciţiul 4.21. În cazul de faţă nu se mai folosesc instrucţiuni de atribuire pentru a da valori iniţiale elementelor tabloului *mon* ci ele se iniţializează prin declaraţia tabloului *mon*.

PROGRAMUL BVI2

```
#include <stdio.h>
#include <stdlib.h>

main() /* exprima o suma de lei in bancnote si monede de
          10000 lei, 5000 lei, 1000 lei etc. */
{
 int
 mon[]={1,5,10,20,50,100,200,500,1000,5000,10000};
 int i=sizeof mon/sizeof(int) -1;
 long s,q;
 char t[255];

 for ( ; ; ) {
     printf("suma= ");
     if(gets(t) == NULL ) {
        printf("s-a tastat EOF\n");
        exit(1);
     }
     if(sscanf(t,"%ld",&s) == 1 && s > 0 )
        break;
     printf("nu s-a tastat un intreg pozitiv\n");
 } /* sfirsit for */
 do {
     q=s/mon[i];
```

```
        if(q)
            if(i<6) /* monede */
                if( q == 1 ) /* o moneda */
                    if(i==0) /* o moneda de 1 leu */
                        printf("o moneda de 1 leu\n");
                    else
                        printf("o moneda a %d lei\n",
                                mon[i]);
                else /* mai multe monede */
                    if(i==0) /* monede de 1 leu */
                        printf("%ld monede de 1 leu\n",
                                q);
                    else /*  monede a mon[i] lei */
                        printf("%ld monede a %d lei\n",
                               q,mon[i]);
            else /* bancnote */
                if(q==1) /* o bancnota */
                    printf("o bancnota a %d lei\n",
                            mon[i]);
                else /* mai multe bancnote */
                    printf("%ld bancnote a %d lei\n",
                           q,mon[i]);
    } while ( s %= mon[i--]);
}
```

Observaţie:

La declararea lui *i* s-a făcut iniţializare cu ajutorul expresiei:

sizeof mon / sizeof(int) -1

Această expresie are ca valoarea numărul elementelor tabloului *mon* micşorat cu 1, adică indicele ultimului element al tabloului. Într-adevăr, *sizeof mon* are ca valoare dimensiunea, în octeţi, a zonei de memorie alocată tabloului *mon*, iar *sizeof(int)* are ca valoare numărul octeţilor necesari pentru a reprezenta o dată de tip *int*. Raportul lor ne dă chiar numărul elementelor tabloului *mon*.

6.3 Să se scrie o funcţie care afişează elementele unui tablou de tip *char* şi citeşte un întreg de tip *int*.

Funcţia are parametrul *text* care este un tablou unidimensional de tip *char*. Ea afişează şirul de caractere păstrat în tabloul *text* înainte de a

citi întregul. Întregul citit se atribuie variabilei globale *v_int*. Funcţia returnează valoarea zero la întîlnirea sfîrşitului de fişier şi 1 în caz contrar. La eroare, se reia citirea întregului după retastarea acestuia de către operator.

FUNCŢIA BVI3

```
int cit_int(char text[])
/* - citeste un intreg si-l atribuie variabilei globale v-int;
   - returneaza:
          0 - la intilnirea sfirsitului de fisier;
          1 - altfel. */
{
 char t[255];
 char texter[]="nu s-a tastat un intreg\n";
 extern int v_int;

 for ( ; ; ) {
     printf(text);
     if(gets(t) == NULL)
        return 0;
     if(sscanf(t,"%d",&v_int) == 1)
        return 1;
     printf(texter);
 }
}
```

6.4 Să se scrie o funcţie care citeşte un întreg de tip *int* care aparţine unui interval dat.

Funcţia are parametri:

text - Tablou unidimensional de tip *char* care are aceeaşi semnificaţie ca în exerciţiul precedent.

inf - Întreg de tip *int* care reprezintă limita inferioară a intervalului căreia trebuie să-i aparţină numărul citit.

sup - Întreg de tip *int* care reprezintă limita superioară a aceluiaşi interval.

Funcţia atribuie întregul citit variabilei globale *v_int*. Ea returnează valoarea zero la întîlnirea sfîrşitului de fişier şi 1 în caz contrar.

```
int cit_int(char []); /* prototip */
int cit_int_lim(char text[], int inf,int sup)
/* - citeste un intreg de tip int ce apartine intervalului [inf,sup] si-l
     atribuie variabilei v_int;
   - returneaza:
          0 - la intilnirea sfirsitului de fisier;
          1 - altfel. */
{
 extern int v_int;

 for( ; ; ) {
     if(cit_int(text) == 0)
        return 0; /* s-a intilnit EOF */
     if(v_int >= inf && v_int <= sup)
        return 1;
     printf("intregul tastat nu apartine\
           intervalului:");
     printf("[%d,%d]\n", inf,sup);
     printf("se reia citirea\n");
 }
}
```

6.5 Să se scrie o funcţie care validează o dată calendaristică.

Funcţia are 3 parametri de tip *int*:

zi - Numărul zilei din cadrul lunii.
luna - Numărul lunii.
an - Anul calendaristic.

Funcţia returnează:

1 - Dacă data calendaristică este validă.
0 - Altfel.

În exerciţiile din această carte se presupune că anul calendaristic aparţine intervalului [1600,4900]. În acest caz, anul este bisect dacă expresia:

an%4==0 && an%100!=0 || an%400==0

are valoarea adevărat (vezi exerciţiul 3.12.).

Funcţia utilizează un tablou global de tip *int* ale cărui elemente au ca valori numărul zilelor din lunile calendaristice. Numele acestui tablou este *nrzile*.

nrzile[i] are ca valoare numărul zilelor din luna a i-a (i=1,2,...,12).

FUNCŢIA BVI5

```
int v_calend(int zi,int luna,int an)
/*- valideaza data calendaristica;
  - returneaza:
        1 - daca este corecta;
        0 - altfel. */
{
 extern int nrzile[];

 if(an < 1600 || an > 4900 ) {
    printf("anul nu este in intervalul\
          [1600,4900]");
    return 0;
 }
 if(luna <1 || luna > 12 ) {
    printf("luna=%d eronata\n", luna);
    return 0;
 }
 if(zi < 1 || zi > nrzile[luna] +
      (luna == 2 && ( an%4 == 0 &&
       an%100 || an%400 == 0 )))
 {
    printf("ziua=%d eronata\n",zi);
    return 0;
 }
 return 1; /* data calendaristica valida */
}
```

Observaţie:

Elementul *nrzile[2]* are ca valoare numărul zilelor din luna februarie pentru un an nebisect, deci 28.

Dacă anul este bisect, atunci expresia:

(1) an%4==0 && an%100 | | an%400==0

are valoarea 1. În acest caz numărul zilelor din luna februarie se măreşte cu 1. Aceasta nu este valabil şi pentru celelalte luni. Deci

nrzile[luna]

se măreşte cu 1 cînd expresia (1) are valoarea 1 şi cînd expresia

(2) luna==2

este adevărată. Cu alte cuvinte, numărul zilelor din februarie se mărește cu 1, cînd atît expresie (1), cît și expresie (2) sînt adevărate și numai atunci. Aceasta înseamnă că expresia:

(2) && (1)

are valoarea 1 atunci și numai atunci cînd anul este bisect și luna este februarie.

6.6 Să se scrie o funcție care determină dintr-o dată calendaristică de forma zi, lună și an, numărul zilei din an pentru data respectivă.

Funcția returnează numărul determinat din parametri *zi*, *luna* și *an*.

Dacă, de exemplu, data calendaristică este 1 martie 1994, atunci ea reprezintă ziua 60 din anul respectiv. Într-adevăr, pînă la 1 martie 1994 au trecut lunile ianuarie și februarie, deci împreună cu 1 martie sînt:

31+28+1=60 zile

Într-un an bisect data de 1 martie este ziua 61 din anul respectiv.

Procesul de calcul constă din însumarea zilelor din lunile precedente lunii calendaristice definită de *luna*, la ziua curentă definită de zi.

Funcția de față presupune că data calendaristică este validă.

FUNCȚIA BVI6

```
int zi_din_an(int zi,int luna,int an)
/* determina ziua din an si returneaza numarul astfel determinat */
{
 extern int nrzile[];
 int bisect=an%4 == 0 && an%100 || an%400 == 0;
 int i;

 for( i=1; i < luna; i++ )
    zi += nrzile[i] + (i == 2 && bisect);
 return zi;
}
```

6.7 Să se scrie o funcție care dintr-o dată calendaristică definită prin numărul zilei din an și anul respectiv, determină luna și ziua din luna respectivă.

Această funcție este inversa funcției *zi_din_an* definită în exercițiul

precedent. Ea definește două valori:

numărul lunii

și

numărul zilei din luna respectivă.

Funcția poate fi realizată în una din variantele:

- Să existe un parametru formal care să fie un tablou de tip *int*. Fie acesta *tzll* și care să corespundă unui parametru efectiv care să fie un tablou de 2 elemente de tip *int*. În acest caz se poate atribui, de exemplu, lui tzll[0] numărul zilei din lună și lui tzll[1] numărul lunii.
- Să existe două variabile globale la care să li se atribuie cele două valori: ziua din lună și respectiv luna.

Funcția realizată mai jos utilizează parametrul *tzll*.

FUNCȚIA BVI7

```
void luna_si_ziua ( int zz, int an,int tzll[])
/* - determina luna si ziua din luna;
        zz - ziua din an;
        an - anul calendaristic;
   - la revenire:
        tzll[0] are ca valoare ziua din luna;
        tzll[1] are ca valoare luna calendaristica */
{
 int bisect=an%4==0 && an%100 || an%400 ==0;
 int i;
 extern int nrzile[];

 for(i=1; zz > nrzile[i] + (i==2 && bisect);i++)
    zz -=(nrzile[i] + ( i==2 && bisect));
 tzll[0] = zz;
 tzll[1] = i;
}
```

6.8 Să se scrie o funcție care citește o dată calendaristică compusă din zi, luna și an.

Funcția validează data calendaristică respectivă. Ea returnează valoarea:

0	- La întîlnirea sfîrșitului de fișier.
1	- Altfel (data calendaristică corectă).

Are un parametru *tzzllaa*, care este un tablou de tip *int*. La revenire, elementele acestui tablou au valorile:

tzzllaa[0] - Ziua citită.
tzzllaa[1] - Luna citită.
tzzllaa[2] - Anul citit.

FUNCŢIA BVI8

```
int cit_data_calend( int tzzllaa[])
/* - citeste o data calendaristica, o valideaza si o pastreaza in
     tabloul tzzllaa:
          tzzllaa[0] - ziua;
          tzzllaa[1] - luna;
          tzzllaa[2] - anul;
   - functia returneaza:
          0 - la intilnirea sfirsitului de fisier;
          1 - altfel */
{
 extern int v_int;
 static char ziua[] = "ziua: ";
 static char luna[] = "luna: ";
 static char anul[] = "anul: ";
 static char eroare[]= "s-a tastat EOF";

 for( ; ; ) {

/* se citeste ziua */
     if(cit_int_lim(ziua,1,31) == 0) {
        printf("%s\n",eroare);
        return 0;
     }
     tzzllaa[0] = v_int;

/* se citeste luna */
     if(cit_int_lim(luna,1,12) == 0 ) {
        printf("%s\n",eroare);
        return 0;
     }
     tzzllaa[1] = v_int;

/* se citeste anul */
     if(cit_int_lim(anul,1600,4900) == 0 ) {
```

```
        printf("%s\n",eroare);
        return 0;
    }
    tzzllaa[2] = v_int;
```

```
/* validare data calendaristica */
    if(v_calend(tzzllaa[0],tzzllaa[1],tzzllaa[2]))
        return 1;
    printf("data calendaristica este eronata\n");
    printf("se reia citirea datei\
          calendaristice\n");
  }
}
```

6.9 Să se scrie un program care citeşte o dată calendaristică şi afişează data calendaristică pentru ziua următoare.

Programul se realizează conform următorilor paşi:

1. Se citeşte şi se validează data calendaristică prin apelul funcţiei *v_calend*.
2. Dacă data citită este 31 decembrie dintr-un anumit an, atunci ziua următoare este 1 ianuarie din anul următor, apoi se trece la pasul 6.

   Altfel se trece la pasul 3.
3. Se determină ziua din an (funcţia *zi_din_an*).
4. Se incrementează ziua din an.
5. Se determină luna şi ziua din lună (funcţia *luna_si_ziua*).
6. Se afişează data calendaristică a zilei următoare.

PROGRAMUL BVI9

```
#include <stdio.h>
#include <stdlib.h>
#include "bvi3.cpp"  /* functia cit_int */
#include "bvi4.cpp"  /* functia cit_int_lim */
#include "bvi5.cpp"  /* functia v_calend */
#include "bvi6.cpp"  /* functia zi_din_an */
#include "bvi7.cpp"  /* functia luna_si_ziua */
#include "bvi8.cpp"  /* functia cit_data_calend */

/* variabile globale */
```

```
int v_int; /* se utilizeaza in functiile:
                 cit_int;
                 cit_int_lim;
                 cit_data_calend. */

int nrzile[]={0,31,28,31,30,31,30,31,31,30,
              31,30,31};
/* se utilizeaza in functiile:
    v_calend;
    zi_din_an;
    luna_si_ziua */

main() /* citeste o data calendaristica, o valideaza si in caz ca este
          corecta, afiseaza data calendaristica a zilei urmatoare */
{
 int data_calend[3];
 int zl[2];

/* citeste si valideaza data calendaristica */
 if(cit_data_calend(data_calend) == 0)
    exit(1);
 if(data_calend[0] == 31 && data_calend[1] == 12 )
 {

/* 31 decembrie; ziua urmatoare este 1 ianuarie din anul urmator */
    zl[0] = 1; /* 1 */
    zl[1] = 1; /* ianuarie */
    data_calend[2]++; /* anul urmator */
 }
 else /* se determina ziua urmatoare */
    luna_si_ziua(zi_din_an(data_calend[0],
                 data_calend[1],data_calend[2])+1,
                 data_calend[2],zl);

/* afiseaza data calendaristica a zilei urmatoare */
 printf("ziua:%d\tluna:%d\tanul:%d\n",
        zl[0],zl[1],data_calend[2]);
}
```

7. PROGRAMARE MODULARĂ

Un program poate fi format din mai multe *module*.

Numim *modul sursă*, o parte a textului sursă al programului care se compilează printr-o singură compilare şi separat de restul textului sursă al programului respectiv. Rezultatul compilării unui modul sursă este un *modul obiect*.

În cele ce urmează, prin *modul* vom înţelege un modul sursă. Modulele obiect le vom mai numi şi module de *tip OBJ*.

De obicei, programele simple se compun dintr-un singur modul sau un număr relativ mic de module (4-5).

În componenţa unui modul intră proceduri "înrudite". Această noţiune nu este definită riguros. De obicei, spunem că mai multe proceduri sînt înrudite dacă utilizează în comun diferite date. De exemplu, funcţiile *zi_din_an* şi *luna_si_ziua*, definite în capitolul precedent, se consideră că sînt înrudite. Ele utilizează în comun tabloul global *nrzile*.

Un modul se compune din unul sau mai multe fişiere. În cazul în care el se compune din mai multe fişiere, acestea se compilează împreună incluzîndu-le unul în altul cu ajutorul construcţiei #include.

Modulele de tip OBJ, obţinute prin compilările modulelor din compunerea unui program, pot fi reunite într-un fişier executabil (cu extensia .EXE) cu ajutorul editorului de legături.

Modulele unui program se definesc ca rezultat al procesului de *descompunere* a unei probleme în *subprobleme mai simple*. Acest proces de descompunere se poate continua, adică subproblemele obţinute la o primă descompunere pot fi şi ele la rîndul lor descompuse în altele mai simple şi aşa mai departe, pînă cînd se ajunge ca toate subproblemele de la nivelele inferioare să fie relativ simple. Diferite componente ale unei astfel de descompuneri se realizează prin module. De obicei, aceste module se pot realiza în paralel şi relativ independent de către mai mulţi programatori, ceea ce conduce la creşterea eficienţei în programare.

Un avantaj mare pe care îl oferă modulele, este aşa numita posibilitate de "ascundere a datelor". Aceasta înseamnă că la o parte sau chiar la toate datele unui modul au acces direct numai procedurile din compunerea modulului respectiv. Adesea se spune că datele respective sînt *vizibile* numai la nivel de modul. Ele pot fi utilizate în alte module numai prin intermediul procedurilor modulului în care sînt vizibile. Mai mult decît atît, pot exista cazuri în care este avantajos ca şi anumite proceduri dintr-un modul să fie

ascunse. La acestea nu se pot face apeluri din alte module. Ele pot fi apelate numai de procedurile modulului în care sînt definite şi ascunse.

Facilitatea de "ascundere" a datelor şi procedurilor în module constituie o *protecţie* împotriva deteriorării datelor prin accese neautorizate din alte module. Această protecţie devine importantă mai ales în cazul problemelor complexe, cînd participă colective mari pentru programarea şi implementarea lor.

Prin *programarea modulară* înţelegem stilul de programare care are la bază utilizarea de module în vederea "ascunderii" datelor şi procedurilor pentru a realiza protecţia datelor respective faţă de accese neautorizate.

Limbajul C a fost proiectat în ideea de a permite utilizarea programării modulare. În acest caz modulul conţine funcţii "înrudite". Datele statice, declarate în afara funcţiilor modulului, pot fi utilizate în comun de către aceste funcţii. Ele sînt ascunse în modul, deoarece funcţii din alte module nu pot face acces direct la ele.

De asemenea, se pot defini şi funcţii statice, ceea ce conduce la ascunderea lor în modulul în care au fost definite.

O funcţie statică poate fi apelată numai de funcţii definite în acelaşi modul cu ea.

Exerciţii:

7.1 Să se realizeze un modul care implementează o stivă ale cărei elemente sînt de tip *int*.

Prin *stivă* înţelegem o mulţime ordonată de elemente la care se are acces în conformitate cu principiul *LIFO* (*Last In First Out*).

Cel mai simplu procedeu de implementare a unei stive este păstrarea elementelor ei într-un tablou unidimensional. Ulterior vom vedea şi alte moduri de implementare a unei stive.

În zona de memorie alocată stivei se pot păstra elementele ei, unul după altul. De asemenea, ele se pot scoate din zona de memorie respectivă, unul cîte unul, în ordine inversă păstrării lor.

Ultimul element pus pe stivă se spune că este în *vîrful* stivei. Primul element pus pe stivă se află la *baza* stivei.

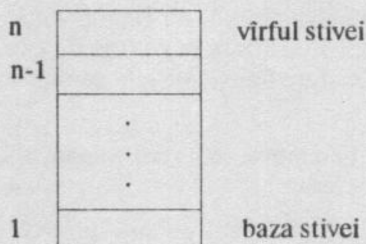
Vîrful stivei se modifică atunci cînd se pune un element pe stivă sau cînd se scoate de pe stivă. În primul caz *lungimea* stivei (numărul elementelor din stivă) creşte, iar în cel de al doilea caz descreşte.

Într-adevăr, cînd se pune un element pe stivă, atunci acesta se pune după cel aflat în vîrful stivei şi prin aceasta vîrful stivei se modifică aşa încît

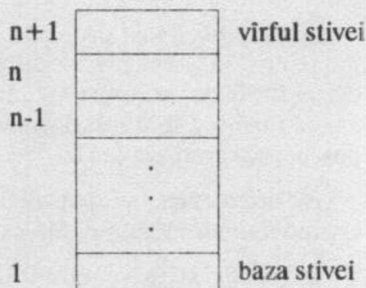
elementul nou pus pe stivă să fie în vîrful ei. Cînd se ia un element de pe stivă, atunci acesta este cel aflat în vîrful stivei şi apoi vîrful stivei se modifică aşa încît, elementul care a fost pus pe stivă înaintea celui scos, să fie în vîrful ei.

Punerea unui element pe stivă

Înainte de a pune un element pe stivă

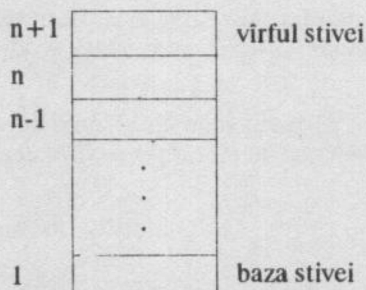


După ce s-a pus un element pe stivă

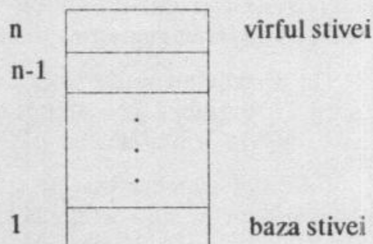


Scoaterea unui element de pe stivă

Înainte de a scoate un element de pe stivă



După ce s-a scos un element de pe stivă



Se observă că totdeauna, ultimul element pus pe stivă este primul care se scoate din stivă. De aici provine afirmaţia că o stivă se gestionează conform principiului *LIFO*.

Pentru a implementa o stivă vor fi necesare două funcţii:

- una care să pună un element pe stivă

şi

- una care să scoată un element din stivă.

Aceste funcţii au denumiri deja consacrate în literatura de specialitate şi anume, funcţia care pune un element pe stivă se numeşte *push*, iar cealaltă *pop*.

Vom păstra şi noi aceste denumiri.

Am spus mai sus că vom implementa stiva folosind un tablou unidimensional. Acesta va fi în cazul de faţă de tip *int*. Numim *stack* acest tablou.

stack[0] este elementul de la baza stivei.

Funcţiile *push* şi *pop* gestionează vîrful stivei şi de aceea ele au nevoie de o variabilă care să definească indicele elementului din vîrful stivei. De obicei, se păstrează indicele *primului element liber*, adică indicele elementului tabloului *stack* care urmează imediat după elementul aflat în vîrful stivei (vezi [2]).

Numim *next* variabila a cărei valoare curentă este primul element liber al tabloului *stack*.

Evident iniţial next=0, deoarece la început nefiind nici un element pe stivă, stack[0] este primul element liber.

Utilizatorul stivei nu trebuie să aibă acces direct la elementele stivei şi de aceea atît tabloul *stack*, cît şi variabila *next* trebuie "ascunse" în modul. În felul acesta, utilizatorul are posibilitatea să pună un element pe stivă în vîrful ei apelînd funcţia *push* şi să scoată elementul din vîrful stivei apelînd funcţia *pop*. Rezultă că aceste două funcţii nu se "ascund" în modul.

De obicei, se definesc şi alte funcţii (vezi[5]) cum ar fi:

clear	- Pentru a vida stiva.
empty	- Stabileşte dacă stiva este vidă sau nu.
full	- Stabileşte dacă stiva este plină sau nu.
top	- Permite acces la elementul din vîrful stivei, fără a-l scoate din stivă.

Toate aceste funcţii trebuie să aibă acces atît la *stack*, cît şi la variabila *next*.

De aceea, *stack* şi *next* se vor declara statice în afara corpurilor lor.

Cele 6 funcţii amintite mai sus vor avea un caracter global, ele putînd fi apelate din orice modul al programului.

Funcţia *push* are prototipul:

void push(int *x*);

În principiu, ea pune pe stivă valoarea lui *x*, deci trebuie să facă atribuirea:

stack[next]=x

şi apoi să incrementeze pe *next* pentru ca aceasta să definească locul liber curent. De aceea, atribuirea de mai sus se poate realiza împreună cu incrementarea lui *next*:

stack[next++]=x.

Funcţia trebuie să testeze, în prealabil, dacă există loc liber pentru a păstra pe *x* şi numai în acest caz se va executa atribuirea de mai sus. În caz că stiva este plină (depăşită), se dă un mesaj de eroare.

Funcţia *pop* are prototipul:

int pop(void);

Ea returnează valoarea din vîrful stivei. În acest scop se va decrementa *next* şi apoi se va returna valoarea elementului

stack[next].

Aceste două acţiuni se pot realiza cu ajutorul instrucţiunii:

return stack[--next];

Se observă că prin aceasta vîrful stivei coboară cu un element, deoarece *next*, care exprimă primul element liber, este chiar indicele elementului eliminat de pe stivă.

Înainte de a executa instrucţiunea de mai sus, funcţia *pop* va testa dacă stiva nu cumva este vidă. În cazul în care stiva este vidă, funcţia *pop* va afişa un mesaj de eroare şi va returna valoarea zero.

Funcţia *top* este ca şi *pop*, cu deosebirea că nu se elimină elementul din vîrful stivei, ci numai se returnează valoarea lui.

Funcţia *clear* are prototipul:

void clear(void);

Ea videază stiva, ceea ce se realizează simplu prin atribuirea valorii zero variabilei *next*. În felul acesta, tot tabloul devine liber.

Funcţia *empty* are prototipul:

int empty(void);

Ea returnează valoarea 1 dacă stiva este vidă şi zero în caz contrar.

Funcţia *full* are prototipul:

int full(void);

Ea returnează valoarea 1 dacă stiva este plină şi zero în caz contrar.

Aceste funcţii, împreună cu declaraţiile lui *stack* şi *next* se păstrează într-un fişier care formează modulul ce implementează stiva *stack* prin intermediul unui tablou de tip *int*.

MODULUL BVII1

```
#define MAX 1000

static int stack[MAX];
static int next = 0;

void push( int x) /* pune x pe stiva */
{
 if(next < MAX )
     stack[next++] = x;
 else
     printf("stiva este plina\n");
}

int pop() /* scoate din stiva elementul din virful ei */
{
 if( next > 0 )
     return stack[--next];
 else
     printf("stiva vida\n");
}

int top() /* returneaza elementul din virful stivei */
{
 if(next > 0 )
     return stack[next-1];
 else
     printf("stiva vida\n");
}
```

```
void clear() /* videaza stiva */
{
 next = 0;
}

int empty() /* returneaza 1 daca stiva este vida si 0 altfel */
{
 return !next;
}

int full() /* returneaza 1 daca stiva este plina si 0 altfel */
{
 return next == MAX;
}
```

7.2 Să se scrie un program care transcrie o expresie cu operanzi numere naturale, în formă poloneză postfixată.

Prin expresie aritmetică înţelegem o expresie în care pot fi utilizate numai cele 4 operaţii binare: adunare (+), scădere (-), înmulţire (*) şi împărţire (/).

Se pot utiliza parantezele rotunde pentru a schimba prioritatea operatorilor.

Forma *poloneză postfixată* a unei expresii aritmetice se poate defini ca mai jos:

- forma poloneză postfixată a unui operand coincide cu el însuşi dacă nu este inclus în paranteze rotunde;
- forma poloneză postfixată a unui operand de forma

  (exp)

  coincide cu forma poloneză postfixată a lui *exp*;
- dacă operatorul binar *op* leagă expresiile *fexp1* şi *fexp2* care sînt în formă poloneză postfixată:

  fexp1 op fexp2

  atunci forma poloneză postfixată a acestei expresii este

  fexp1 fexp2 op

Exemple:

Expresie aritmetică	Echivalentul ei în formă poloneză postfixată
1+2	1 2 +
1+2*3	1 2 3 * +
(1+2)*3	1 2 + 3 *

Un algoritm simplu pentru a transforma o expresie aritmetică în formă poloneză postfixată se bazează pe utilizarea unei stive în care se păstrează temporar operatorii expresiei. În aces scop se va folosi stiva implementată prin modulul din exemplul precedent.

Expresia aritmetică în formă poloneză postfixată, pe măsură ce se construieşte, se păstrează într-un tablou *tefpp* de tip caracter.

Algoritmul de traducere are următorii paşi:

a. Un operand se păstrează automat în tabloul *tefpp*. Acest lucru rezultă din faptul că prin această transformare operanzii îşi păstrează ordinea din expresia iniţială. Apoi se continuă cu următorul element al expresiei.

b. Paranteza deschisă se introduce automat în stivă.

c. Un operator se introduce în stivă dacă în vîrful stivei se află:

   - paranteza deschisă;
   - un operator de prioritate mai mică;

   sau

   - stiva este vidă.

   Dacă în vîrful stivei se află un operator de prioritate mai mare sau egală cu a celui curent, atunci operatorul din vîrful stivei se scoate din stivă şi se păstrează în tabloul *tefpp*. Apoi se compară din nou operatorul din vîrful stivei cu cel curent şi se reia pasul c.

d. La întîlnirea unei paranteze închise, se scot pe rînd operatorii din stivă şi se introduc în tabloul *tefpp*, pînă la întîlnirea parantezei deschise din stivă. Aceasta pur şi simplu se elimină din stivă. Apoi se continuă cu următorul element al expresiei.

e. La terminarea parcurgerii expresiei, se scot pe rînd toţi operatorii care se mai află în stivă şi se trec în *tefpp*.

Se presupune că expresia se termină cu punct şi virgulă.

La baza stivei se va afla caracterul NUL ('\0').

Programul de faţă poate transforma mai multe expresii, pînă la tastarea sfîrşitului de fişier.

Menţionăm că metoda de transcriere a expresiilor descrisă mai sus, funcţionează exact numai dacă expresia respectivă este corectă din punct de vedere sintactic.

În mod normal acest program se compune din două module, unul care a fost definit în exerciţiul precedent (fişierul sursă BVII1.CPP) şi prin care se implementează o stivă pe care se pun elementele de tip *int*, iar cel de al doilea modul defineşte funcţia principală.

Cele două module se pot compila distinct obţinîndu-se două module de tip OBJ care urmează apoi a fi link-editate împreună pentru a obţine imaginea executabilă a programului.

Cele două fişiere de tip sursă pot fi compilate şi împreună, procedînd ca pînă acum, adică incluzînd fişerul BVII1.CPP în fişierul BVII2.CPP.

De asemenea, fişierele respective pot fi compilate şi link- editate folosind un fişier de tip *Project*. Acesta se editează utilizînd meniul *Project* al mediului integrat de dezvoltare Turbo C++.

Mai jos s-a adoptat varianta cu includerea fişierului BVII1.CPP.

Fişierul BVII2.CPP, alături de funcţia principală mai conţine o funcţie denumită *sca*. Aceasta are prototipul:

int *sca*(**int** *c*);

Ea realizează saltul peste caracterele albe.

Dacă *c* are ca valoare codul ASCII al unui caracter alb, atunci se citesc caracterele care urmează, pînă la întîlnirea primului caracter care nu este alb. La revenire, se returnează codul ultimului caracter citit. Dacă *c* are ca valoare codul unui caracter ASCII care nu este alb, atunci se revine din funcţie cu codul caracterului respectiv.

PROGRAMUL BVII2

```
#include <stdio.h>
#include "bvii1.cpp"

#define MAXTEFPP 1000

int sca(int); /* prototipul functiei sca */
```

```
/* prototipul functiilor push, pop si clear sînt necesare cind cele doua
   fisiere se compileaza separat; in cazul de fata sînt in plus */
void push(int);
int pop(void);
void clear(void);

main() /* transforma expresii aritmetice
          in forma poloneza postfixata */
{
 int c,i,j;
 char tefpp[MAXTEFPP];

 for ( c=getchar(); c != EOF; ) {
     i=0; /* indice pentru tefpp */
     clear(); /* videaza stiva */
     push('\0'); /* zero la baza stivei */
     while((c=sca(c)) != ';' && c != EOF ) {

/* 1 */

/* c nu este alb, punct si virgula sau sfirsitul de fisier */
        while(c>='0' && c<='9') { /* 2 */

 /* se pastreaza un operand */
            tefpp[i++] = c;
            c = getchar();
        } /* sfirsit while 2 */

/* s-a terminat un operand; poate urma un caracter alb, un operator,
   punct si virgula sau EOF */
        c = sca(c) ; /* salt peste caracterele albe daca exista */
        tefpp[i++] = ' '; /* spatiu dupa operand */
        switch(c) {
            case '(': /* paranteza deschisa se pune pe stiva */
                push(c);
                break;
            case ')': /* paranteza inchisa: se scot operatorii
                         din stiva pina la paranteza deschisa
                         si se trec in tefpp */
                while((c=pop()) != '(')
                    tefpp[i++] = c;
                break;
```

```
            case '+':
            case '-': /* se scot operatorii de pe stiva si se trec in
                             tabloul tefpp */
               while((j=pop())=='+'||
                      j=='-'||j=='*'||j=='/')
                  tefpp[i++] = j;

/* ultimul element scos de pe stiva se repune */
               push(j);

/* se pune pe stiva operatorul curent */
               push(c);
               break;

            case '*':
            case '/': /* se scot de pe stiva operatorii * si / */
               while((j=pop()) == '*' || j == '/' )
                  tefpp[i++] = j;

/* ultimul element scos de pe stiva se repune */
               push(j);

/* se pune pe stiva operatorul curent */
               push(c);
               break;

            case ';': /* sfirsit expresie */
               break;

            default:
/* expresie eronata */
               printf("caracter eronat:%c\t%d\n",
                       c,c);

 * se cauta sfirsitul expresiei sau EOF */
               while( c!= ';' && c != EOF )
                  c=getchar();
         } /* sfirsit switch */
         if( c!=';' && c != EOF )

/* avans la caracterul urmator din expresie */
            c = getchar();
      } /* sfirsit while 1 */
```

```
/* sfirsit expresie: se trec operatorii din stiva in tefpp,
   daca exista */
        while((j = pop()) != '\0')
            tefpp[i++] = j;

/* se afiseaza forma poloneza postfixata a expresiei curente */
        putchar('\n');
        for(j=0; j<i; j++)
            putchar(tefpp[j]);
        putchar('\n');

/* la EOF se termina executia; altfel se citeste primul caracter al
   expresiei urmatoare */
        if(c == EOF )
            break;
        c = getchar();
        c = sca(c); /* avans peste caractere albe */

  } /* sfirsit for neimbricat */
} /* sfirsit main */

int sca(int x) /* salt peste caractere albe */
{
  while(x == ' ' || x == '\t' || x == '\n')
     x = getchar();
  return x;
}
```

8. POINTERI

Un *pointer* este o variabilă care are ca valori adrese. Pointerii se utilizează pentru a face referire la date cunoscute prin adresele lor. Astfel, dacă *p* este o variabilă de tip pointer care are ca valoare adresa lui *x*, atunci

**p*

reprezintă chiar valoarea lui *x*.

Fie de exemplu:

int x,y;

atunci dacă *p* are ca valoare adresa lui *x*, atribuirea:

y=x+100

este identică cu:

y=*p+100

În mod analog, atribuirea:

x=3

este identică cu:

*p=3

În construcţia **p* utilizată mai sus, caracterul * se consideră ca fiind un operator unar care furnizează valoarea din zona de memorie a cărei adresă este conţinută în *p*.

Operatorul unar * are aceeaşi prioritate ca şi ceilalţi operatori unari din limbajul C şi se asociază de la dreapta spre stînga (vezi 3.2.16.).

Dacă *p* conţine adresa zonei de memorie alocată lui *x*, vom spune că *p* *pointează* spre *x*.

De asemenea, dacă *p* are ca valoare adresa de început a unei zone de memorie care conţine o dată de tipul *tip*, atunci vom spune că *p* pointează spre *tip*.

Menţionăm că în legătură cu noţiunea de *pointer*, în limba română se utilizează şi alte denumiri, ca de exemplu:

- referinţă;
- localizator;
- reper;

- indicator de adresă etc.

În cartea de faţă păstrăm denumirea englezească.

Pentru a atribui o adresă unei variabile de tip *pointer* se poate folosi operatorul unar & (vezi 3.2.11.). Astfel, dacă dorim ca *p* să pointeze spre *x* (să aibă ca valoare adresa lui *x*), atunci putem utiliza atribuirea:

p=&*x*

Operatorul unar & este numit operator *adresă* sau de *referenţiere*. Operatorul unar * îl vom numi operator de *indirectare* sau de *dereferenţiere*.

Ultima denumire a operatorului unar * decurge din efectul invers al acestuia faţă de operatorul unar &. Într-adevăr, expresia:

*&*x*

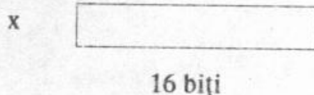
are aceeaşi valoare ca şi operandul *x*.

Exemplu:

Fie declaraţia:

int x;

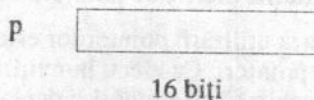
Variabilei *x* i se alocă o zonă de memorie de 16 biţi:



16 biţi

Fie 1000 adresa zonei alocate variabilei *x*.

Dacă *p* este o variabilă pointer, atunci ei i se atribuie,de asemenea, o zonă de memorie de 16 biţi (o să vedem ulterior că uneori această zonă va fi de 32 biţi):

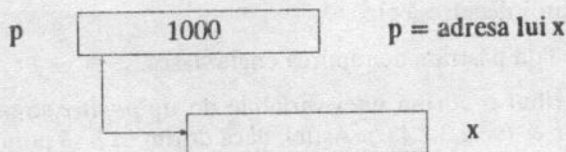


16 biţi

Folosind instrucţiunea de atribuire:

p=&*x*;

lui *p* i se atribuie valoarea 1000:



Instrucţiunea:

x = 10;

atribuie lui *x* valoarea 10:



Acelaşi efect se obţine folosind instrucţiunea:

*p = 10;

În concluzie, instrucţiunea:

x = 10;

este echivalentă cu secvenţa:

```
p = &x;
*p = x;
```

sau

```
p = &x, *p = x;
```

Noţiunea de pointer joacă un rol important deoarece permite calcule cu adrese. Acestea sînt utile mai ales în scrierea programelor de sistem. Astfel de calcule sînt proprii limbajelor de asamblare. Introducerea lor în limbajul C, oferă programatorilor posibilitatea de a scrie programe mai optime, decît cele realizate fără a beneficia de facilităţile oferite de pointeri.

Un beneficiu obţinut simplu pe baza utilizării pointerilor este înlocuirea expresiilor cu indici prin expresii cu pointeri. Ca efect, înmulţirile utilizate la evaluarea variabilelor cu indici se schimbă cu adunări şi deplasări. Un alt beneficiu obţinut cu ajutorul pointerilor este posibilitatea alocării dinamice a memoriei prin alte mijloace decît cel utilizat la alocarea datelor automatice.

Folosirea funcţiilor ca parametri este şi el un beneficiu al utilizării pointerilor.

În capitolul de față se descriu aspectele de bază cu privire la utilizarea pointerilor în scrierea programelor C.

8.1. Declarația de pointer și tipul pointer

Un *pointer* se declară ca orice variabilă, cu singura deosebire că numele este precedat de caracterul *. Astfel, dacă dorim să declarăm variabila *p* utilizată mai sus pentru a păstra adresa lui *x*, vom folosi declarația:

```
int *p;
```

Tipul *int* stabilește faptul că *p* conține adrese de zone de memorie în care se păstrează date de tip *int*. Declarația de mai sus se poate interpreta astfel:

> **p* reprezintă conținutul zonei de memorie spre care pointează *p*, iar acest conținut are tipul *int*.

În general, un pointer se declară prin:

*tip *nume;*

ceea ce înseamnă că *nume* este un pointer care pointează spre o zonă de memorie ce conține o dată de tipul *tip*.

Comparînd declarația de mai sus cu cea obișnuită:

tip nume;

putem considera că:

tip *

dintr-o declarație de pointeri reprezintă

tip

dintr-o declarație obișnuită. De aceea, construcția:

tip *

se spune că reprezintă un tip nou, tipul *pointer*. Acest tip se spune că este tipul *pointer spre tip*.

Dacă avem declarațiile:

```
int x;
int *p;
float y;
```

atunci atribuirea

```
p = &x
```

este corectă, în timp ce

```
p = &y
```

nu este corectă, deoarece *p* poate conţine numai adrese de zone de memorie în care se păstrează date de tip *int*.

Dacă există declaraţia:

```
float *q;
```

atunci se poate folosi atribuirea

```
q = &y
```

Exemple:

```
int x,y;
int *p;
```

1.

```
y = x+100;
```

este echivalentă cu secvenţa:

```
p = &x;
y = *p+100;
```

2.

```
x = y;
```

este echivalentă cu secvenţa:

```
p = &x;
*p = y;
```

3.

```
x++;
```

este echivalentă cu secvenţa:

```
p = &x;
(*p)++;
```

Există cazuri în care dorim ca un pointer să fie utilizat cu mai multe tipuri de date. În acest caz, la declararea lui nu putem specifica un tip, ca în exemplele de mai sus. Aceasta se realizează folosind cuvîntul *void*:

```
void *nume;
```

Exemplu:

```
int x;
```

```
float y;
char c;
void *p;
...
p = &x;
...
p = &y;
...
p = &c;
...
```

Deoarece *p* a fost declarat cu ajutorul cuvîntului cheie *void*, lui *p* i se pot atribui adrese de zone de memorie care pot conţine date de tipuri diferite: *int, float, char* etc.

Cînd se folosesc pointeri de tip *void*, este necesar să se facă conversii explicite prin expresii de tip *cast*, pentru a preciza tipul datei spre care pointează un astfel de pointer. Într-adevăr, dacă se utilizează *p* declarat ca mai sus, atunci o atribuire de forma:

```
*p = 10
```

nu este corectă, deoarece nu este definit tipul datei spre care pointează *p*.

Pentru a putea realiza o astfel de atribuire, va trebui să convertim valoarea lui *p* spre tipul "pointer spre tipul *int*". Tipul *pointer spre int* se exprimă prin construcţia:

```
int *
```

Amintim că valoarea unui operand se poate converti spre tipul *tip* folosind operatorul unar *(tip)*:

(tip) operand

De exemplu, pentru a converti valoarea lui *y*, din exemplul de mai sus, spre *long*, vom scrie:

```
(long) y
```

În cazul de faţă, valoarea lui *p* trebuie convertită spre tipul *int* * deci vom folosi expresia cast:

```
(int *) p
```

În felul acesta atribuirea de mai sus devine:

```
*(int *)p = 10
```

În mod analog, o expresie de forma:

```
*p+1.5
```

este eronată. Este necesar să se convertească explicit valoarea lui *p* spre tipul de dată conţinut în zona de memorie a cărui adresă este valoarea lui *p*. Rezultă că expresiile:

```
*(int *)p+1.5
*(float *)p+1.5
*(char *)p+1.5
```

pot fi utilizate în locul expresiei de mai sus. Expresia:

```
*(int *)p+1.5
```

se evaluează astfel:

a. Adresa care este valoarea lui *p* se interpretează ca fiind adresa zonei de memorie care conţine o dată de tip *int*.

b Valoarea de la adresa definită de expresia:

```
(int *)p
```

se converteşte din *int* spre *double* în conformitate cu regula conversiilor implicite.

c. Valoarea obţinută la punctul b se adună cu 1.5 şi suma este de tip *double*.

Conversia tipului pointer:

```
void *
```

spre un tip pointer concret (*int* *, *float* * etc.) este totdeauna posibilă şi nu înseamnă altceva decît *precizarea* tipului de pointer pe care îl are valoarea pointerului la care se aplică conversia respectivă. Deci, conversia realizată prin expresia *cast*:

```
(int *)p
```

precizează faptul că *p* are ca valoare o adresă a unei zone de memorie în care se păstrează întregi de tip *int*.

Utilizarea tipului:

```
void *
```

asigură o flexibilitate mare în utilizarea pointerilor. Cu toate acestea, se recomandă să nu se utilizeze în mod abuziv, deoarece aceasta poate fi şi o sursă de erori. Într-adevăr, programatorul trebuie să ştie, în fiecare moment, ce fel de tip de pointer este valoarea atribuită variabilei pointer de tip *void* *.

De exemplu, dacă se utilizează expresia:

```
(int *)p
```

într-un moment în care *p* are ca valoare adresa unei zone de memorie care conţine o dată flotantă, rezultatul va fi imprevizibil.

8.2. Realizarea apelului prin referinţă utilizînd parametri de tip pointer

Am văzut că în limbajul C apelul este prin valoare. Aceasta înseamnă că la un apel, parametrilor formali li se atribuie valorile parametrilor efectivi care le corespund (vezi 4.15.).

În cazul în care un parametru efectiv este un nume de tablou, apelul prin valoare devine prin referinţă, parametrului formal corespunzător lui i se atribuie ca valoare adresa primului element al tabloului. Deci, dacă se face apelul:

```
int tab[...];
...
f(tab);
...
```

şi *f* are antetul:

```
void f(int v[])
```

atunci la apel *v* are aceeaşi valoare cu *tab*. Aceasta înseamnă că tab[0] şi v[0] exprimă acelaşi lucru, adică valoarea lui tab[0]. În general, tab[i] şi v[i] au aceeaşi valoare şi anume valoarea elementului de indice *i* al tabloului *tab*.

Folosind pointeri şi în alte cazuri, putem să transformăm apelul prin valoare în apel prin referinţă. Astfel, dacă *x* este o variabilă simplă, atunci noi putem să transferăm, la un apel, în locul valorii lui *x*, adresa lui *x*:

```
int x;
...
h(x); /* se transfera valoarea lui x */
...
g(&x); /* se transfera adresa lui x */
...
```

În acest caz cele două funcţii vor avea antete diferite şi anume:

```
void h(int i)
```

şi

```
void g(int *pi)
```

În cazul funcţiei *g* parametrul formal *pi* este pointer spre date de tip *int*.

La apel, lui *pi* i se atribuie ca valoare adresa lui *x*. De aceea, funcţia *g* are posibilitatea să modifice valoarea lui *x*. De exemplu, dacă în corpul funcţiei *g* se utilizează atribuirea:

```
...
*pi=100;
...
```

atunci valoarea 100 se atribuie variabilei *x*, a cărei adresă s-a atribuit la apel lui *pi*.

În felul acesta, apelul prin valoare se poate folosi pentru a realiza apelul prin referinţă.

În principiu, în limbajul C, apelul prin referinţă se poate realiza dacă parametrul efectiv are ca valoare o adresă, iar parametrul formal corespunzător este un *pointer*.

Ulterior o să vedem că în limbajul C++ a fost introdus apelul prin referinţă şi el există împreună cu cel prin valoare.

Exerciţii:

8.1 Să se scrie o funcţie care dintr-o dată calendaristică definită prin numărul zilei din an şi anul respectiv, determină luna şi ziua din luna respectivă.

Această funcţie a fost definită în exerciţiul 6.7. Ea a avut trei parametri:

zz - Ziua din an.
an - Anul.
tzzll - Tablou de tip *int* de 2 elemente.

Funcţia atribuie ziua determinată, elementului tzzll[0], iar luna elementului tzzll[1].

În funcţia de mai jos se înlocuieşte parametrul *tzzll* cu doi parametri de tip pointer.

FUNCŢIA BVIII1

```
void pluna_si_ziua( int zz, int an,int *zi,
                    int *luna)
/* determina luna si ziua din luna;
      zz - ziua din an;
      an - anul;
      zi - pointer a carui valoare este adresa zonei in care
```

```
                    se pastreaza ziua determinata de functie
               luna - pointer a carui valoare este adresa zonei in care
                    se pastreaza luna determinata de functie */
{
 int bisect = an %4 == 0 && an%100 || an%400 == 0;
 int i;
 extern int nrzile[];

 for(i = 1; zz > nrzile[i]+ (i==2 && bisect); i++)
    zz -= (nrzile[i] + (i==2 &&bisect));

 *zi = zz; /* pastreaza valoarea lui zz in zona de memorie alocata
              parametrului efectiv corespunzator parametrului
              formal zi */

 *luna = i; /* pastreaza valoarea lui i in zona de memorie alocata
               parametrului efectiv corespunzator prametrului
               formal luna */
}
```

8.2 Să se scrie o funcţie care afişează caracterele unui tablou şi citeşte un întreg de tip *int*.

Funcţia are doi parametri:

text - Tablou unidimensional de tip caracter şi care are aceeaşi utilizare ca parametrul *text* din funcţia 6.3.

x - Pointer spre întregi de tip *int*; are ca valoare adresa zonei de memorie în care se păstrează valoarea întregului citit.

Această funcţie este asemănătoare cu funcţia definită în exerciţiul 6.3. Diferenţa constă în aceea că, funcţia definită în exerciţiul 6.3. atribuie numărul citit variabilei globale *v_int*, în loc să-l transfere funcţiei care face apelul, ca în cazul de faţă.

Funcţia de faţă returnează aceleaşi valori ca şi cea amintită mai sus, adică 0 la întîlnirea sfîrşitului de fişier şi 1 altfel.

FUNCŢIA BVIII2

```
int pcit_int(char text[], int *x)
/* - citeste un intreg si-l pastreaza in zona de memorie a carei adresa este
     valoarea lui x;
   - returneaza:
```

```
        0 - la intilnirea sfirsitului de fisier;
        1 - altfel */
{
 char t[255];

 for ( ; ; ) {
     printf(text);
     if(gets(t) == NULL)
         return 0;
     if(sscanf(t,"%d", x) == 1)
         return 1;
 }
}
```

Observaţie:

Deoarece *x* are ca valoare chiar adresa zonei de memorie în care se păstrează întregul convertit din ASCII în *int* prin funcţia *sscanf*, în apelul acestei funcţii se foloseşte ca parametru efectiv chiar *x*. El nu mai trebuie să fie precedat de operatorul adresă (&).

8.3 Să se scrie o funcţie care citeşte un întreg de tip *int* care aparţine unui interval dat.

Ea are parametri:

text - Tablou unidimensional de tip *char* care are aceeaşi utilizare ca în funcţia definită în exerciţiul 6.4. (funcţia *cit_int_lim*).

inf - Întreg de tip *int* care are aceeaşi utilizare ca în funcţia *cit_int_lim*.

sup - Întreg de tip *int* care are aceeaşi utilizare ca în funcţia *cit_int_lim*.

pint - Pointer spre întregi de tip *int*; are ca valoare adresa zonei de memorie în care se păstrează numărul citit.

Ca şi funcţia *cit_int_lim*, funcţia de faţă returnează:

0 - La întîlnirea sfirşitului de fişier.

1 - Altfel.

FUNCŢIA BVIII3

```
int pcit_int(char [], int *); /* prototip */
```

```
int pcit_int_lim(char text[], int inf,
                 int sup, int *pint)
/* - citeste un intreg de tip int ce apartine intervalului [inf,sup] si-l
     pastreaza in zona de memorie a carei adresa este valoarea parame-
     trului pint;
   - returneaza:
        0 - la intilnirea sfirsitului de fisier;
        1 - altfel. */
{
 for( ; ; ) {
     if(pcit_int(text, pint) == 0)
        return 0; /* s-a intilnit EOF */
     if(*pint >= inf && *pint <= sup)
        return 1;
     printf("intregul tastat nu apartine\
            intervalului:");
     printf("[%d,%d]\n", inf,sup);
     printf("se reia citirea\n");
  }
}
```

8.4 Să se scrie o funcţie care citeşte o dată calendaristică compusă din *zi*, *luna* şi *an*.

Funcţia validează data calendaristică respectivă. Ea are trei parametri de tip pointer:

pzi - Are ca valoare adresa zonei de memorie în care se păstrează numărul zilei.

pluna - Are ca valoare adresa zonei de memorie în care se păstrează numărul lunii.

pan - Are ca valoare adresa zonei de memorie în care se păstrează anul.

Funcţia returnează:

0 - Dacă s-a întîlnit sfîrşitul de fişier.

1 - Altfel.

Pentru validarea datei calendaristice se apelează funcţia *v_calend* definită în exerciţiul 6.5.

Funcţia de faţă este analogă cu cea definită în exerciţiul 6.8 Diferenţa constă în utilizarea parametrilor şi a funcţiei care citeşte cele 3 valori din compunerea datei calendaristice. Astfel, funcţia de faţă utilizează trei

parametri de tip pointer spre *int*, pentru transferul celor trei valori citite, spre deosebire de funcţia *cit_data_calend* definită în exerciţiul 6.8., care utilizează în acelaşi scop un tablou de tip *int*. De asemenea, funcţia de faţă apelează funcţia *pcit_int_lim* pentru a citi cele trei valori ale datei calendaristice, spre deosebire de funcţia *cit_data_calend* care utilizează funcţia *cit_int_lim*.

FUNCŢIA BVIII4

```
int pcit_int_lim( char [], int,
                  int, int *); /* prototip */

int pcit_data_calend( int *pzi,int *pluna,int *pan)
/* - citeste o data calendaristica, o valideaza si o pastreaza in zonele de
     memorie a caror adrese sînt definite de cei trei parametri formali
     de tip pointer;
   - functia returneaza:
        0 - la intilnirea sfirsitului de fisier;
        1 - altfel. */
{
 static char ziua[] = "ziua: ";
 static char luna[] = "luna: ";
 static char an[] = "anul: ";
 static char er[]= "s-a tastat EOF";

 for( ; ; ) {
/* se citeste ziua */
    if(pcit_int_lim(ziua,1,31,pzi) == 0) {
        printf("%s\n",er);
        return 0;
    }

/* se citeste luna */
    if(pcit_int_lim(luna,1,12,pluna) == 0 ) {
        printf("%s\n",er);
        return 0;
    }

/* se citeste anul */
    if(pcit_int_lim(an,1600,4900,pan) == 0 ) {
        printf("%s\n",er);
        return 0;
```

```
}

/* validare data calendaristica */
     if(v_calend(*pzi, *pluna, *pan))
        return 1;
     printf("data calendaristica este eronata\n");
     printf("se reia citirea datei\
            calendaristice\n");
   }
}
```

8.5 Să se scrie un program care citeşte o dată calendaristică şi afişează data calendaristică pentru ziua următoare.

Acest program este analog cu cel definit în exerciţiul 6.9. Programul de faţă utilizează funcţii ce a i ca parametri pointeri, spre deosebire de cel definit în exerciţiul 6.9., care apelează funcţii ce utilizează variabila globală *v_int* şi parametrii de tip tablou.

PROGRAMUL BVIII5

```
#include <stdio.h>
#include <stdlib.h>
#include "bviii2.cpp" /* functia pcit_int */
#include "bviii3.cpp" /* functia pcit_int_lim */
#include "bvi5.cpp" /* functia v_calend */
#include "bvi6.cpp" /* functia zi_din_an */
#include "bviii1.cpp" /* functia pluna_si_ziua */
#include "bviii4.cpp" /* functia pcit_data_calend */

int nrzile[]={0,31,28,31,30,31,30,31,
              31,30,31,30,31};

main() /* citeste o data calendaristica, o valideaza si in caz ca este
          corecta, afiseaza data calendaristica a zilei urmatoare */
{
 int zz,ll,aa;

/* citeste si valideaza data calendaristica */
 if(pcit_data_calend(&zz,&ll,&aa) == 0)
    exit(1);
 if(zz == 31 && ll == 12 ) {
```

```
/* 31 decembrie; ziua urmatoare este 1 ianuarie din anul urmator */
        zz = 1;
        ll = 1; /* ianuarie */
        aa++; /* anul urmator */
    }
    else /* se determina ziua urmatoare */
        pluna_si_ziua(zi_din_an(zz,ll,aa)+1,
                      aa,&zz,&ll);

/* afiseaza data calendaristica a zilei urmatoare */
  printf("ziua:%d\tluna:%d\tanul:%d\n",zz,ll,aa);
}
```

Observaţie:

Parametrii efectivi *&zz, &ll, &aa* de la apelul funcţiei *pcit_data_calend* atribuie parametrilor formali corespunzători, adresele zonelor de memorie alocate variabilelor *zz, ll,* şi respectiv *aa*.

La revenirea din funcţia *pcit_data_calend* zonele de memorie alocate acestor variabile vor conţine valorile corespunzătoare zilei, lunii şi anului care au fost citite prin apelul respectiv (dacă nu s-a tastat sfîrşitul de fişier).

8.6 Să se scrie o funcţie care citeşte:

- valoarea variabilei *m* de tip *int*;
- valoarea variabilei *n* de tip *int*;
- m*n numere care reprezintă elementele unei matrice de ordinul m*n.

Funcţia are următorii parametri:

dmat - Tablou unidimensional de tip *double* în care se păstrează elementele matricei prin liniarizare.

max - Maximul produsului *m*n* admis.

nrlin - Pointer a cărui valoare este adresa zonei de memorie în care se păstrează valoarea lui *m*.

nrcol - Pointer a cărui valoare este adresa zonei de memorie în care se păstrează valoarea lui *n*.

Această funcţie este asemănătoare cu funcţia *gdcitmat* definită în exerciţiul 5.1. Diferenţa dintre ele constă în faptul că funcţia *gdcitmat* nu utilizează parametri de tip pointer pentru a transfera valorile lui *m* şi *n* în

afara funcţiei, ci două variabile globale.

FUNCŢIA BVIII6

```
int ndcit(int, double []); /* prototip */

int pdcitmat(double dmat[],int max,
             int *nrlin, int *nrcol)
/* - citeste pe:
      m-numar de linii;
      n-numar de coloane;
   - m*n numere de tip double pe care le pastreaza in matricea dmat
     prin liniarizare;
   - returneaza valoarea m*n;
        m - se pastreaza in zona de memorie definita de
            parametrul nrlin;
        n - se pastreaza in zona de memorie definita de
            parametrul nrcol. */
{
 int i;
 char t[255];
 char er[]="s-a tastat EOF\n";

 do { /* se citesc valorile lui m si n */
     do { /* citeste pe m */
         printf("numarul de linii= ");
         if(gets(t) == NULL ) {
             printf(er);
             exit(1);
         }
         if(sscanf(t,"%d", nrlin)==1 &&
                  *nrlin > 0 && *nrlin <= max)
             break;
         printf("nu s-a tastat un intreg in\
                intervalul [1,%d]\n", max);
     } while(1);

     do { /* citeste pe n */
         printf("numarul de coloane= ");
         if(gets(t) == NULL ) {
             printf(er);
             exit(1);
         }
```

```
            if(sscanf(t,"%d",nrcol)==1 &&
                        *nrcol > 0 && *nrcol <= max)
                break;
            printf("nu s-a tastat un intreg in\
                    intervalul [1,%d]\n", max);
        } while (1);

        i = *nrlin * *nrcol;
        if(i <=max)
            break;
        printf("produsul m*n=%d depaseste pe\
                max=%d\n", i, max );
        printf("se reiau citirile lui m si n\n");
    } while(1);

   /* se citesc elementele matricei tastate pe linii */
    if(ndcit(i,dmat) != i ) {
        printf("nu s-au tastat %d elemente\n", i);
        exit(1);
    }
    return i;
   }
```

Observaţii:

1. Funcţia *ndcit* este definită în exerciţiul 4.37.
2. Citirea numerelor *m* şi *n* se poate realiza mai simplu folosind funcţia *pcit_int_lim* definită în exerciţiul 8.3.

8.7 Să se scrie un program care citeşte elementele de tip *double* ale două matrice *a* şi *b*, calculează produsul lor şi-l afişează.

Dacă

c=a*b

atunci

c[i,j]=a[i,0]*b[0,j]+a[i,1]*b[1,j]+...+a[i,n-1]*b[n-1,j]

pentru

i=0,1,...,m-1; j=0,1,...,s-1

unde:

a - Este o matrice de ordinul *m*n*.

b - Este o matrice de ordinul *n*s*.

Matricea produs *c* este de ordinul m*s.

PROGRAMUL BVIII7

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include "biv37.cpp" /* functia ndcit */
#include "bviii6.cpp" /* functia pdcitmat */

#define MAX 900

main() /* citeste elementele a doua matrice, calculeaza si afiseaza
          matricea produs, cite 4 elemente pe o linie */
{
 int i,j,k;
 int m,n,p,s,r,q;
 int mn,np;
 double a[MAX],b[MAX],c[MAX];

/* citeste matricea a*/
 mn = pdcitmat(a,MAX, &m, &n);

/* citeste matricea b*/
 np =pdcitmat(b,MAX,&p,&s);
 if(n != p) {
     printf("numarul coloanelor matricei\
           a = %d \n",n);
     printf("difera de numarul liniilor ");
     printf("matricei b = %d \n",p);
     exit(1);
 }

/* se realizeaza produsul c = a*b */
 for(i=0; i < m; i++) {
     q = i*n;
     for(j=0; j < s; j++) {
         r = i*s + j;
         c[r] = 0.0;
         for(k=0; k < n; k++ )
             c[r] += a[q+k]* b[k*s+j];
```

```
        }
    }

  /* afiseaza elementele matricei produs */
   printf("\n\n\t\tmatricea produs\n");
   k=1;
   for(i=0;i<m;i++) {
       p = i*s;
       for(j=0;j<s;j++) {
           printf("c[%d,%d]=%8g ",i,j,c[p+j]);
           if(j%4 == 3) {

  /* afiseaza 4 elemente pe un rind */
               printf("\n");
               k++; /* numara rindurile */
           }
           if(k==23) {
               printf("actionati o tasta pentru a\
                      continua\n");
               getch();
               k=1;
           }
       }
       printf("\n");
       k++;
   }
}
```

8.3. Legătura dintre pointeri şi tablouri

Numele unui tablou este un *pointer* deoarece el are ca valoare adresa primului său element. Totuşi există o diferenţă între numele unui tablou şi o variabilă de tip pointer. Unei variabile de tip pointer i se atribuie valori la execuţie, în timp ce aceasta nu este posibil să se realizeze pentru numele unui tablou. Acesta tot timpul are ca valoare adresa primului său element. De aceea, se obişnuieşte să se spună că numele unui tablou este un *pointer constant*.

Exemplu:

```
int t[10];
int *p;
int x;
...
```

```
p=t; /* in urma acestei atribuiri p are aceeasi valoare ca si t,
        adica adresa lui t[0] */
x=t[0];
```

şi

```
x=*p;
```

au acelaşi efect: atribuie lui *x* valoarea elementului t[0].

Un parametru formal ce corespunde unui parametru efectiv care este un nume de tablou unidimensional, poate fi declarat fie ca *tablou*, fie ca *pointer* spre tipul tabloului.

Fie declaraţia:

```
int tab[10];
```

şi apelul:

```
f(tab);
```

Funcţia *f* poate avea unul din următoarele antete:

```
void f(int t[])
```

sau

```
void f(int *t)
```

Într-adevăr, declaraţia:

```
int t[]
```

defineşte pe *t* ca nume de tablou, iar parametrului *t* i se atribuie la apel valoarea lui *tab*, adică adresa lui tab[0]. În felul acesta, construcţiile tab[i] şi t[i] reprezintă unul şi acelaşi element al i+1 - lea al tabloului *tab*.

Parametrul formal *t* declarat în acest fel este deci un pointer (are ca valoare o adresă) spre *int*. Spre deosebire de *tab*, *t* este chiar un pointer variabil. El este alocat pe stivă şi lui i se atribuie o valoare prin apelul funcţiei. Ulterior o să vedem că noi putem să modificăm valoarea lui *t*. Nu acelaşi lucru se poate spune despre *tab*. Acesta este un pointer constant, a cărui valoare nu poate fi schimbată la execuţie.

Din cele de mai sus rezultă că parametrul *t* este un pointer variabil spre *int* şi deci el poate fi declarat printr-o declaraţie de forma:

```
int *t
```

În concluzie, dacă un parametru formal corespunde unui parametru efectiv care este nume de tablou unidimensional, atunci el poate fi declarat fie ca tablou:

tip nume_parametru_formal[]

fie ca pointer:

*tip *nume_parametru_formal*

Cele două declaraţii sînt echivalente şi de aceea, ambele declaraţii permit ca în corpul funcţiei să se utilizeze construcţia:

nume_parametru_formal[indice]

8.4. Operaţii cu pointeri

Asupra pointerilor se pot face diferite operaţii pe care le precizăm în continuare.

8.4.1. Operaţii de incrementare şi decrementare a pointerilor

Operatorii ++ şi -- se pot aplica la operanzi de tip *pointer*. Ei se execută altfel decît asupra datelor care nu sînt pointeri.

Operatorul de incrementare (++) aplicat unui operand de tip pointer spre tipul *t*, măreşte adresa,care este valoarea operandului, cu numărul de octeţi necesari pentru a păstra o dată de tip *t*.

Operatorul de decrementare are un efect similar, diferenţa constînd în aceea că în acest caz valoarea operandului se micşorează cu numărul de octeţi necesari pentru a păstra o dată de tipul spre care pointează operandul.

De exemplu, dacă avem declaraţia:

```
int *p;
```

atunci expresiile:

```
++p
```

şi

```
p++
```

măresc valoarea lui *p* cu 2, deoarece o dată de tip *int* se păstrează pe 2 octeţi.

În mod analog, expresiile:

```
--p
```

şi

p--

micşorează valoarea lui *p* cu 2.

Aceste operaţii sînt utile cînd se au în vedere prelucrări de date de tip tablou.

Exemplu:

```
double tab[10];
double *p;
int i;
```

Fie:

p = &tab[i];

unde 0 < i < n-1. Instrucţiunile expresie:

p++; şi ++p;

măresc valoarea lui *p* cu 8, deci *p* va avea ca valoare adresa următoare elementului tab[i], adică adresa elementului tab[i+1].

În mod analog, instrucţiunile expresie:

p--; şi --p;

micşorează valoarea lui *p* cu 8, deci *p* va avea ca valoare adresa elementului precedent lui tab[i], adică adresa elementului

tab[i-1].

8.4.2. Adunarea şi scăderea unui întreg dintr-un pointer

Dacă *p* este un pointer spre tipul *t* şi *n* un întreg, atunci se pot utiliza expresiile:

p+n

şi

p-n

Expresia:

p+n

are ca valoare, valoarea lui *p* mărită cu produsul r*n, unde prin *r* am notat numărul de octeţi necesari pentru a păstra în memorie o dată de tipul *t*.

În mod analog, valoarea expresiei

p-n

este valoarea lui *p* micşorată cu produsul r*n.

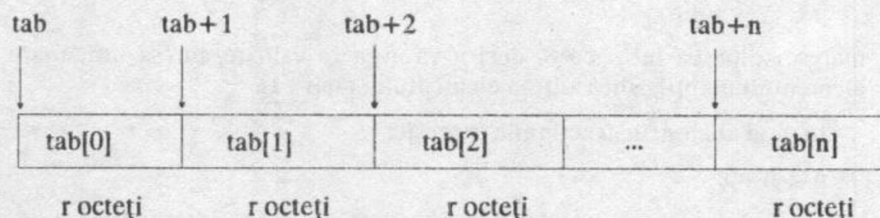
Fie *tab* un tablou unidimensional de tip *t*. Atunci *tab* este un pointer (constant) şi deci o expresie de forma:

tab+n

este corectă. Conform celor spuse mai sus, rezultă că această expresie este chiar adresa elementului tab[n]. Într-adevăr, *tab* este adresa lui tab[0]. Atunci tab+1 are ca valoare adresa lui tab[0] mărită cu *r*, *r* fiind numărul de octeţi alocaţi pentru o dată de tip *t*, adică numărul de octeţi ocupaţi de elementul tab[0].

Deci tab+1 are ca valoare adresa elementului tab[1].

În general, tab+n va avea ca valoare adresa elementului tab[n].



Deoarece tab+n este adresa elementului tab[n], rezultă că

*(tab+n)

are ca valoare chiar valoarea elementului tab[n]. Deci, o atribuire de forma:

x = tab[n]

este echivalentă cu:

x = *(tab+n)

În general, *variabilele cu indici* se pot înlocui prin *expresii cu pointeri*. Astfel de înlocuiri conduc la optimizări înlocuindu-se operaţiile de înmulţire care intervin la evaluarea variabilelor cu indici, prin adunări. De aceea, se recomandă utilizarea expresiilor cu pointeri în locul variabilelor cu indici.

8.4.3. Compararea a doi pointeri

Doi pointeri care pointează spre elementele aceluiaşi tablou pot fi comparaţi folosind operatorii de relaţie şi de egalitate.

Astfel, dacă *p* pointează spre elementul t[i] al tabloului *t* (adică are ca valoare adresa elementului t[i]) şi *q* spre elementul t[j] al aceluiaşi tablou, atunci expresiile:

p < q, p <= q, p >=q, p > q, p==q şi p != q

sînt legale. De exemplu, expresia:

p < q

are valoarea *adevărat* (1), dacă i < j şi *fals* (0) în caz contrar.

În mod analog se evaluează şi celelalte expresii de mai sus.

Operatorii de egalitate (== şi !=) pot fi folosiţi pentru a compara pointerii cu o constantă specială *NULL*. Aceasta este definită în fişierul *stdio.h* astfel:

#define NULL 0

Ea reprezintă aşa numitul *pointer nul*.

Dacă *p* este un pointer spre orice tip, atunci se pot face comparaţii de forma:

p == NULL

şi

p != NULL

În limbajul C++ se recomandă să nu se utilizeze constanta *NULL*, ci chiar valoarea ei zero:

p == 0

şi

p !=0

sau echivalentele lor:

!p

şi

p

Într-adevăr, p == 0 are valoarea adevărat atunci şi numai atunci cînd !p are valoarea adevărat.

Acelaşi lucru se poate spune şi despre celelalte două expresii:

p != 0

are valoarea fals atunci şi numai atunci cînd *p* are valoarea zero, adică cînd *p* este fals.

În general, dacă un pointer are valoarea zero (pointerul nul), înseamnă că el nu defineşte o adresă. Aşa cum o să vedem mai tîrziu, astfel de teste sînt uneori necesare pentru a pune în evidenţă anumite erori.

Ţinînd seama de faptul că *NULL* nu se mai utilizează în programarea curentă în limbajul C++, nu o vom utiliza nici la scrierea programelor în C. Pînă în prezent am utilizat constanta *NULL* la apelul funcţiei *gets*. Aceasta returnează pointerul spre zona în care se păstrează şirul de caractere citit. Deoarece la întîlnirea sfîrşitului de fişier, conţinutul acestei zone nu este definit, funcţia *gets* returnează zero în acest caz.

8.4.4. Diferenţa a doi pointeri

Doi pointeri care pointează spre elementele aceluiaşi tablou pot fi scăzuţi. Fie *p* un pointer spre elementul t[i] al tabloului *t* şi *q* un pointer spre elementul t[i+n] al aceluiaşi tablou. Atunci diferenţa:

q-t

are valoarea *n*.

Exerciţii:

8.8 Să se scrie o funcţie care citeşte cel mult *n* elemente de tip *double* şi le păstrează în zona de memorie a cărei adresă de început este valoarea parametrului formal *p* al funcţiei. Funcţia returnează numărul numerelor citite.

Funcţia de faţă este analogă cu funcţia *ndcit* definită în exerciţiul 4.37. În acest caz se utilizează expresii cu pointeri în locul indicilor.

FUNCŢIA BVIII8

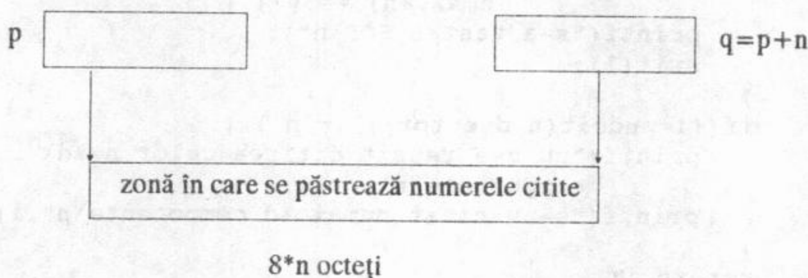
```
int pndcit(int n, double *p)
/* - citeste cel mult n numere si le pastreaza in zona spre care pointeaza
     p;
   - returneaza numarul numerelor citite. */
{
 double d;
 char t[255];
 int i = 0;
 double *q = p+n;

 while( p < q ) {
```

```
        printf("elementul [%d]= ",i);
        if(gets(t) == NULL )
            return i;
        if(sscanf(t,"%lf", &d) != 1)
            break;
        *p++ = d;
        i++;
    }
    return i;
}
```

Observaţii:

1. Parametrul *p* are ca valoare la apel, adresa de început a zonei în care se păstrează numerele citite. Dimensiunea acestei zone este de *8*n* octeţi, pentru a putea păstra cel mult *n* numere de tip *double*.



8*n octeţi

2. Instrucţiunea:

   *p++=d;

   memorează numărul citit în zona spre care pointează *p*.

Totodată pointerul *p* se incrementează, deci valoarea lui se măreşte cu 8. În felul acesta, la iteraţia următoare, *p* pointează spre zona în care urmează a fi memorat elementul următor.

După păstrarea a *n* numere, *p* devine egal cu valoarea lui *q*. Deci ciclul *while* în acest moment trebuie să se termine, dacă nu cumva s-a terminat înainte, deoarece s-au tastat mai puţin de *n* numere.

8.9 Să se scrie o funcţie care citeşte componentele unui vector precedate

de numărul lor. Funcţia returnează numărul componentelor vectorului respectiv.

Numărul componentelor se citeşte apelînd funcţia *pcit_int_lim* definită în exerciţiul 8.3.

Componentele vectorului se citesc apelînd funcţia *pndcit* definită în exerciţiul 8.8.

FUNCŢIA BVIII9

```
int pdvcit(int nmax, double dvector[])
/* - citeste un intreg n si cele n componente ale vectorului dvector;
   - returneaza valoarea lui n. */
{
 int i,n;
 char t[255];

 if(pcit_int_lim("numarul componentelor=",1,
                  nmax,&n) == 0 ) {
     printf("s-a tastat EOF\n");
     exit(1);
 }
 if((i=pndcit(n,dvector)) != n ) {
     printf("nu s-a reusit citirea celor n=%d\
            componente\n", n );
     printf("s-au citit numai %d componente\n",i);
 }
 return i;
}
```

Observaţie:

Funcţia *pcit_int_lim* a fost apelată folosind şirul de caractere:

"numarul componentelor="

ca primul parametru efectiv al apelului. Acest parametru este păstrat de compilator într-o zonă specială rezervată şirului de caractere, iar la apel, se atribuie parametrului formal corespunzător, adresa de început a zonei în care se păstrează şirul respectiv.

8.10 Să se scrie un program care citeşte componentele vectorilor *x* şi *y*, calculează şi afişează produsul lor scalar.

Componentele celor doi vectori sînt precedate de numărul lor.

Programul este similar cu cel definit în exercițiul 4.39. În cazul de față se folosesc expresii cu pointeri în locul variabilelor cu indici.

PROGRAMUL BVIII10

```
#include <stdio.h>
#include <stdlib.h>

#include "bviii2.cpp" /* pcit_int */
#include "bviii3.cpp" /* pcit_int_lim */
#include "bviii8.cpp" /* pndcit */
#include "bviii9.cpp" /* pdvcit */

#define MAX 1000

main() /* - citeste componentele vectorilor x si y precedate de
              numarul lor;
            - calculeaza si afiseaza produsul lor scalar. */
{
 int i,n;
 double *p,*q;
 double x[MAX],y[MAX],s;

/* citeste pe n si componentele vectorului x */
 n = pdvcit(MAX,x);

/* citeste componentele vectorului y */
 if(pndcit(n,y) != n ) {
     printf("nu sint n=%d componente pentru\
             y\n",n);
     exit(1);
 }

/* calculeaza produsul scalar */
 s = 0.0;
 for(p=x,q=y, i=0; i < n; i++,p++,q++)
     s += *p * *q;
 printf("(x,y)= %g\n", s);
}
```

8.5. Modificatorul const

Am văzut în capitolul 1 că o constantă se poate defini prin caracterele care o compun.

Într-adevăr, caracterele din compunerea unei constante definesc atît tipul cît şi valoarea acesteia.

Exemple:

1234	constantă de tip *int*
1.5	constantă de tip *double*
'a'	constantă caracter
"a"	constantă şir

Tot în capitolul 1 s-au definit *constantele simbolice*. O astfel de constantă se defineşte cu ajutorul construcţiei *#define* tratată de preprocesor. Ea are următorul format:

#define *nume succesiune_de_caractere*

Printr-o astfel de construcţie se atribuie un nume unei constante.

La preprocesare, *nume* se înlocuieşte peste tot prin *succesiune_de_caractere*, exceptînd cazurile cînd *nume* se află într-un comentariu sau şir de caractere.

La compilare, *nume* definit printr-o construcţie #define nici nu există în afara şirurilor de caractere şi a comentariilor. De aceea, valoarea lui nici nu poate fi modificată la execuţia programului.

În limbajele C şi C++ există posibilitatea de a defini date constante folosind modificatorul *const* în declaraţii. Printr-o astfel de declaraţie, unui nume i se poate atribui o valoare inţială care nu poate fi modificată, în mod obişnuit, printr-o expresie de atribuire, ca în cazul variabilelor. De aceea, o dată declarată cu ajutorul modificatorului *const* se consideră că este o *constantă*.

Formatele posibile ale unei declaraţii cu modificatorul *const* sînt:

tip **const** *nume = valoare;*
tip **const** **nume = valoare;*
const *tip nume = valoare;*
const *tip *nume;*
const *nume = valoare;*

O declaraţie de forma celor de mai sus se spune că este o *declaraţie de constantă*.

O declaraţie de constantă de forma:

(1) *tip* **const** *nume = valoare;*

este asemănătoare cu o declarţie de variabilă obişnuită de forma:

(2) *tip nume = valoare;*

Diferenţa dintre cele două declaraţii constă în aceea că valoarea lui *nume* atribuită prin declaraţia (1) nu poate fi schimbată folosind o expresie de atribuire de forma:

nume = expresie

în timp ce aceasta este posibil pentru *nume* declarat prin declaraţia (2).

Exemple:

1. `int const i = 10;`

i este o constantă care are valoarea 10. O expresie de atribuire de forma

i = 3

este eronată.

Constanta *i* declarată ca mai sus diferă faţă de o constantă definită prin *#define*, deoarece ea nu este prelucrată de preprocesor. Numele ei există la compilare.

O expresie de forma:

y = i+7

este corectă şi va atribui lui y valoarea 17.

2. `double const pi = 3.14159265;`

   declară numele *pi* ca fiind o constantă de tip *double* şi care are valoarea 3.14159265.

O atribuire de forma:

pi = 3.1415

generează o eroare la compilare.

pi poate fi utilizat în expresii de forma:

```
pi*r*r
sin(pi/2+x)
cos(x-pi)
```

3. `char *const s = "sir";`

Aici *tip* este char *, deci *s* este un *pointer constant* spre zona în care se păstrează şirul de caractere format din literele *s, i, r* şi caracterul *NUL*. În acest caz valoarea lui *s* nu poate fi schimbată. El fiind pointer, are ca valoare o adresă a unei zone de memorie de dimensiune egală cu 4 octeţi. În această zonă se păstrează şirul de caractere indicat mai sus. Conţinutul acestei zone poate fi modificat. Astfel, în timp ce o atribuire de forma s = t; unde *t* este un pointer spre caractere, nu este acceptată de compilator, atribuirile:

```
*s = '1';
*(s+1) = '2';
```

sînt corecte. Cu ajutorul lor se schimbă caracterul *s* cu 1 şi *i* cu 2 în zona spre care pointează *s*.

Declaraţia:

(3) *tip* **const** ***nume** = *valoare;*

defineşte pe *nume* ca un pointer spre o zonă constantă. În acest caz, valoarea pointerului *nume* se poate schimba. De exemplu, dacă se consideră declaraţia:

```
char const *s = "sir";
```

atunci atribuirea:

```
s = t;
```

unde *t* este un pointer spre *char* este corectă. În schimb atribuirile:

```
*s = 'a'
*(s+1) = 'b'
```

sînt eronate, deoarece *s* pointează spre o zonă în care se păstrează o dată constantă. Cu toate acestea, constanta respectivă poate fi modificată folosind un pointer diferit de *s*:

```
char *p;

p= (char *)s; /* p, ca si s, pointeaza spre zona in care se pastreaza
                 sirul de caractere "sir" */
*p = 'a';
*(p+1) = 'b';
```

Aceste atribuiri sînt corecte, deoarece *p* nu mai este un pointer spre o zonă constantă.

Declaraţia:

(4) const *tip nume = valoare;*

este identică cu declaraţia (1) dacă *tip* nu este un tip pointer.

Dacă *tip* este un tip pointer, adică declaraţia (4) este de forma:

(5) **const** *tip *nume = valoare;*

atunci ea este identică cu declaraţia (3) de mai sus. De obicei, se utilizează formatul (5), în locul formatului (3) pentru a declara un pointer spre o zonă constantă (a cărei valoare nu poate fi modificată direct, adică prin atribuiri în care se foloseşte *nume*:

*nume = ...
*(nume+k)=...).

Declaraţia

(6) **const** *tip *nume*

se utilizează pentru a declara un parametru formal.

Fie funcţia *f* de antet:

*tip f(tip *nume)*

La apelul funcţiei *f*, parametrului formal *nume* i se atribuie ca valoare o adresă. Am văzut că acest fapt dă posibilitatea funcţiei *f* să modifice *data* păstrată în zona de memorie spre care pointează nume folosind o atribuire de forma:

**nume = valoare*

Există cazuri în care funcţia apelată în acest fel nu are voie să modifice *data* din zona de memorie a cărei adresă este atribuită unui parametru formal. Ea trebuie numai să aibă acces la *data* respectivă. Pentru a proteja *data* faţă de eventualele atribuiri neautorizate, vom declara parametrul formal respectiv ca un pointer spre o dată constantă. În acest caz, antetul funcţiei *f* devine:

*tip f(***const** *tip *nume)*

În acest caz o atribuire de forma:

**nume = valoare*

este interzisă. De aceea, declaraţia:

const *tip *nume*

se recomandă a fi utilizată pentru orice parametru formal care la apel are ca valoare adresa unei zone de memorie al cărui conţinut nu poate fi modificat de funcţia apelată.

8.6. Funcţii standard utilizate la prelucrarea şirurilor de caractere

Biblioteca standard conţine o serie de funcţii care permit operaţii cu şiruri de caractere. Majoritatea acestor funcţii au prototipul în fişierul *string.h*. Mai jos prezentăm funcţiile mai importante din această clasă.

Un şir de caractere se păstrează într-o zonă de memorie organizată ca tablou unidimensional de tip *char*. Fiecare caracter se păstrează pe cîte un octet prin codul său numeric.

Cel mai frecvent cod utilizat în acest scop este codul ASCII.

După ultimul caracter al şirului se păstrează caracterul *NUL* ('\0').

Pentru a opera cu un şir de caractere se poate utiliza *numele tabloului* ale cărui elemente au ca valori codurile caracterelor şirului respectiv.

Spunem despre acest *nume* că este un pointer constant spre şirul respectiv.

Evident, se pot utiliza şi pointeri variabili spre un şir de caractere.

Exemplu:

char tab[] = "Acesta este un sir";

Şirul de caractere "Acesta este un sir" se păstrează în zona de memorie alocată lui *tab*.

tab are ca valoare adresa de început a zonei de memorie în care se păstrează caracterele şirului.

tab	- Adresa caracterului *A*.
tab+1	- Adresa caracterului *c*.
tab+2	- Adresa caracterului *e*.

etc.

tab[0]	- Codul ASCII al caracterului *A*.
tab[1]	- Codul ASCII al caracterului *c*.

etc.

**tab*	- Codul ASCII al caracterului *A*.
**(tab+1)*	- Codul ASCII al caracterului c.

etc.

Un efect similar se obţine cu ajutorul declaraţiei:

char *const p = "Acesta este un sir";

Şirul de caractere se păstrează într-o zonă de memorie rezervată pentru a păstra şiruri de caractere.

Adresa de început a zonei în care se păstrează şirul de faţă se atribuie pointerului *p*. Acesta, ca şi *tab*, este un pointer constant.

p	- Adresa caracterului *A*.
p+1	- Adresa caracterului *c*.
p+2	- Adresa caracterului *e*.

etc.

*p[0] sau *p*	- Codul ASCII al caracterului *A*.
*p[1] sau *(p+1)*	- Codul ASCII al caracterului *c*.

etc.

În legătură cu şirurile de caractere se au în vedere operaţii de următorul fel:

- calculul lungimii unui şir de caractere;
- copierea şirurilor de caractere;
- concatenarea şirurilor de caractere;
- compararea şirurilor de caractere.

Funcţiile standard prin care se realizează aceste operaţii au fiecare un nume care începe cu prefixul *str* (prescurtare de la *string*).

8.6.1. Lungimea unui şir de caractere

Lungimea unui şir de caractere se defineşte prin numărul de caractere proprii care intră în compunerea şirului respectiv. Caracterul *NUL* este un caracter impropriu şi el nu este considerat la determinarea lungimii unui şir de caractere. Prezenţa lui este însă necesară, deoarece la determinarea lungimii unui şir se numără caracterele acestuia pînă la întîlnirea caracterului *NUL*.

Funcţia pentru determinarea lungimii unui şir de caractere are prototipul:

unsigned strlen(const char ***s**);

Exemple:

1.

```
char *const p = "Acesta este un sir";
```

```
unsigned n;
...
n = strlen(p);
```

Lui *n* i se atribuie valoarea 18 (numărul caracterelor proprii din compunerea şirului spre care pointează *p*).

2.

```
char tab[] = "Acesta este un sir";
int n;
n = strlen(tab);
```

Variabila *n* primeşte aceeaşi valoare ca în exemplul precedent.

3.

```
int n;
n = strlen("Acesta este un sir");
```

Lui *n* i se atribuie aceeaşi valoare ca în exemplele precedente.

Observaţie:

Parametrul formal al funcţiei *strlen* este un pointer spre o dată constantă deoarece funcţia *strlen* nu are voie să modifice caracterele şirului pentru care determină lungimea.

8.6.2. Copierea unui şir de caractere

Adesea este nevoie să se copieze un şir de caractere din zona de memorie în care se află, într-o altă zonă.

În acest scop se poate folosi funcţia de prototip:

char *strcpy(char ****dest*, const char ****sursa***);**

Funcţia copiază şirul de caractere spre care pointează *sursa* în zona de memorie a cărei adresă de început este valoarea lui *dest*.

Funcţia copiază atît caracterele proprii şirului, cît şi caracterul *NUL* de la sfîrşitul şirului respectiv.

Se presupune că zona de memorie în care se face copierea este destul de mare pentru a putea păstra caracterele copiate. În caz contrar se alterează datele păstrate imediat după zona rezervată la adresa definită de parametrul *dest*.

La revenire, funcţia returnează adresa de început a zonei în care s-a

transferat şirul, adică chiar valoarea lui *dest*. Această valoare este pointer spre caractere deci tipul returnat de funcţie este:

char *

De aceea, prototipul funcţiei *strcpy* începe prin construcţia *char* * (tipul valorii returnate de funcţie este pointer spre *char*).

Se observă că parametrul *sursa*, care defineşte zona în care se află şirul ce se copiază, este declarat prin modificatorul *const*. Aceasta deoarece funcţia *strcpy* nu are voie să modifice şirul care se copiază. În schimb, parametrul *dest* nu este declarat cu modificatorul *const* deoarece funcţia *strcpy* modifică zona spre care pointează *dest* (în ea se copiază caracterele şirului).

Exemple:

1.

```
char tab[] = "Acest sir se copiaza";
char t[sizeof tab]; /* are acelasi numar de elemente ca si tab */
...
strcpy(t,tab); /* sirul pastrat in tab se copiaza in zona alocata
                  lui t*/
```

2.

```
char t[100];
strcpy(t,"Acest sir se copiaza");
```

3.

```
char *p = "Acest sir se copiaza";
char t[100];
char *q;
q = strcpy(t,p);
```

Şirul păstrat în zona spre care pointează *p* se transferă în zona spre care pointează *t*. Valoarea lui *t* se atribuie lui *q*.

Pentru a copia cel mult *n* caractere ale unui şir dintr-o zonă de memorie în alta, se va folosi funcţia de prototip:

char *strncpy(char ***_dest_, const char ***_sursa_**, unsigned** *n*);

Dacă *n* > *lungimea şirului* spre care pointează *sursa*, atunci toate caracterele şirului respectiv se transferă în zona spre care pointează *dest*. Altfel se copiază numai primele *n* caractere ale şirului. În rest, funcţia *strncpy* are acelaşi efect ca şi *strcpy*.

Exemplu:

```
char *p = "Acest sir se copiaza truncheat";
char t[10];
strncpy(t,p,sizeof t);
```

8.6.3. Concatenarea şirurilor de caractere

Bibliotecile limbajelor C şi C++ conţin o funcţie care permite concatenarea unui şir de caractere la sfîrşitul unui alt şir de caractere. Una dintre ele are prototipul:

char *strcat(char ***_dest_,const char ***_sursa_**);**

Această funcţie copiază şirul de caractere din zona spre care pointează *sursa*, în zona de memorie care urmează imediat după ultimul caracter propriu al şirului spre care pointează *dest*. Se presupune că zona spre care pointează *dest* este suficientă pentru a păstra caracterele proprii celor două şiruri care se concatenează, plus caracterul *NUL* care termină şirul rezultat în urma concatenării.

Funcţia returnează valoarea lui *dest*.

Exemplu:

```
char tab1[100] = "Limbajul C++";
char tab2[] = "este c incrementat";
strcat(tab1," ");  /* concateneaza caracterul spatiu dupa cel
                      de al doilea caracter + pastrat in tabloul
                      tab1 la initializare */
strcat(tab1,tab2); /* concateneaza textul pastrat in tab2 la
                      initializare, dupa spatiul concatenat prin
                      instructiunea precedenta */
```

Observaţie:

Funcţia *strcat*, la fel ca funcţia *strcpy*, nu trebuie să modifice şirul de caractere spre care pointează *sursa*.

O altă funcţie de bibliotecă utilizată la concatenarea de şiruri este funcţia *strncat*.

Ea are prototipul:

char *strncat(char ***_dest_,const char ***_sursa_**,unsigned** *n***);**

În acest caz se concatenează, la sfîrşitul şirului spre care pointează *dest*, cel mult *n* caractere ale şirului spre care pointează *sursa*.

Dacă *n* > lungimea şirului spre care pointează *sursa*, atunci se concatenează întregul şir, altfel numai primele *n* caractere ale acestuia.

Exemplu:

```
char tabl[100] = "Limbajul E este mai bun decit ";
char tab2[] =
    "limbajul C++ care este un superset a lui C";
strncat(tabl,tab2,12);
```

După revenirea din funcţie, tabloul *tab1* conţine succesiunea de caractere:

Limbajul E este mai bun decit limbajul C++

8.6.4. Compararea şirurilor de caractere

Şirurile de caractere se pot compara folosind codurile ASCII ale caracterelor din compunerea lor.

Fie *s1* şi *s2* două tablouri unidimensionale de tip caracter folosite pentru a păstra, fiecare, cîte un şir de caractere.

Şirurile păstrate în aceste tablouri sînt *egale* dacă au lungimi egale şi s1[i]=s2[i] pentru toate valorile lui *i*.

Şirul păstrat în tabloul *s1* este *mai mic* decît cel păstrat în *s2*, dacă există un indice *i*, aşa incît:

s1[i] < s2[i]

şi

s1[i] = s2[j] pentru j=0,1,...,i-1.

Şirul păstrat în tabloul *s1* este *mai mare* decît cel păstrat în *s2*, dacă există un indice *i*, aşa încît:

s1[i] > s2[i]

şi

s1[j] = s2[j] pentru j=0,1,...,i-1.

Compararea şirurilor de caractere se poate realiza folosind funcţii standard de felul celor de mai jos.

O funcţie utilizată frecvent în compararea şirurilor este funcţia de prototip:

int strcmp(const char ****s1*,const char ****s2***);**

Notăm cu *sir1* şirul de caractere spre care pointează *s1* şi cu *sir2* şirul de caractere spre care pointează *s2* (spunem că un pointer pointează spre un şir dacă valoarea lui este adresa de început a zonei de memorie în care se păstrează şirul respectiv).

Funcţia *strcmp* returnează:

- o valoare negativă dacă sir1 < sir2;
- zero dacă sir1 = sir2;
- o valoare pozitivă dacă sir1 > sir2.

O altă funcţie pentru compararea a două şiruri este funcţia de prototip:

int strncmp(const char ****s1*,const char ****s2***,unsigned** ***n***);**

Această funcţie compară cele două şiruri spre care pointează *s1* şi *s2* utilizînd cel mult primele *n* caractere din fiecare şir. În cazul în care minimul dintre lungimile celor două şiruri este mai mic decît *n*, funcţia *strncmp* realizează aceeaşi comparaţie ca şi funcţia *strcmp*.

Adesea, la compararea şirurilor de caractere dorim să nu se facă distincţie între literele mici şi mari. Acest lucru este posibil dacă se foloseşte funcţia de prototip:

int stricmp(const char ****s1*,const char ****s2***);**

Această funcţie returnează aceleaşi valori ca şi funcţia *strcmp*, cu deosebirea că la compararea literelor nu se face distincţie între literele mari şi mici.

Exemple:

```
char *sir1 = "ABC";
char *sir2 = "abc";
int i;
```

Apelul:

i = strcmp(sir1,sir2);

returnează o valoare negativă, deoarece literele mari au coduri ASCII mai mici decît literele mici (*A* are codul 65, iar *a* are codul 97), deci

"ABC" < "abc"

Apelul:

i = stricmp(sir1,sir2);

returnează zero, deoarece ignorîndu-se diferenţa dintre literele mari şi mici, cele două şiruri devin egale.

Pentru a limita compararea a două şiruri de caractere la primele cel mult *n* caractere ale lor, la comparare ignorîndu-se diferenţa dintre literele mici şi mari, se va folosi funcţia de prototip:

int strincmp(const char *_s1_, const char *_s2_, unsigned _n_);

Exerciţii:

8.11 Să se scrie un program care citeşte o succesiune de cuvinte şi-l afişează pe cel mai lung dintre ele.

Prin *cuvînt* înţelegem o succesiune de caractere diferite de caracterele albe.

Vom presupune că un cuvînt nu are mai mult de 100 de caractere. Cuvintele sînt separate prin caractere albe. La sfîrşit se tastează sfîrşitul de fişer pentru a termina succesiunea de cuvinte.

PROGRAMUL BVIII11

```
#include <stdio.h>
#include <string.h>

#define MAX 100

main() /* citeste o succesiune de cuvinte si-l afiseaza pe cel mai lung
          dintre ele */
{
 int max=0,i;
 char cuvint[MAX+1];
 char cuvint_max[MAX+1];

 while(scanf("%100s",cuvint) != EOF )
     if(max < (i=strlen(cuvint))) {
         max = i;
         strcpy(cuvint_max,cuvint);
     }
 if(max)
     printf("%s\n", cuvint_max);
}
```

Observaţii:

1. Cuvîntul citit se păstrează în tabloul *cuvint* de MAX+1 elemente. La citire se utilizează specificatorul de format:

%100s

care permite să se citească cel mult 100 de caractere diferite de cele albe. Funcţia *scanf* păstrează caracterul NUL după ultimul caracter citit. Deci, un cuvînt de 100 de caractere ocupă 101 octeţi.

2. După citirea unui cuvînt se apelează funcţia *strlen* pentru a determina numărul caracterelor citite prin *scanf* şi păstrate în tabloul *cuvint*.

În acest caz s-a utilizat expresia:

i = strlen(cuvint)

Apoi se compară lungimea cuvîntului citit cu *max*.

Variabila *max* are ca valoare lungimea maximă a cuvintelor citite înaintea celui curent.

Iniţial max = 0, deoarece nu există nici un cuvînt citit.

Dacă *max* este mai mic decît lungimea cuvîntului citit curent, atunci lui *max* i se atribuie această valoare, iar cuvîntul citit este transferat în zona de memorie alocată tabloului *cuvant_max*. În acest scop se apelează funcţia *strcpy*:

strcpy(cuvint_max,cuvint);

8.12 Să se scrie un program care citeşte două cuvinte şi le afişează în ordine crescătoare.

Cuvîntul se defineşte ca în exerciţiul precedent, adică este o succesiune de cel mult 100 de caractere care nu sînt albe. Cuvintele sînt separate prin caractere albe.

PROGRAMUL BVIII12

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define MAX 100

main() /* citeste doua cuvinte si le afiseaza in ordine crescatoare */
{
 char cuv1[MAX+1];
 char cuv2[MAX+1];

 if(scanf("%100s",cuv1) != 1 ) {
```

```
        printf("nu s-a tastat un cuvint\n");
        exit(1);
    }
    if(scanf("%100s",cuv2) != 1 ) {
        printf("nu s-a tastat un cuvint\n");
        exit(1);
    }
    if(strcmp(cuv1,cuv2) < 0 ) {

  /* primul cuvint tastat este mai mic decit cel de-al doilea */
        printf("%s\n",cuv1);
        printf("%s\n",cuv2);
    }
    else {
        printf("%s\n",cuv2);
        printf("%s\n",cuv1);
    }
  }
```

8.13 Să se scrie un program care citeşte o succesiune de cuvinte şi-l afişează pe cel mai mare.

Prin cuvînt înţelegem o succesiune de cel mult 100 de caractere care nu sînt albe. Cuvintele sînt separate prin caractere albe. Succesiunea de cuvinte se termină cu sfîrşitul de fişier.

PROGRAMUL BVIII13

```
  #include <stdio.h>
  #include <string.h>

  #define MAX 100

  main()  /* citeste o succesiune de cuvinte si-l afiseaza pe
             cel mai mare */
  {
    char cuvcrt[MAX+1];
    char cuvmax[MAX+1];

    cuvmax[0] = '\0'; /* cuvmax se initializeaza cu cuvintul vid */
    while(scanf("%100s",cuvcrt) != EOF )
        if(strcmp(cuvcrt,cuvmax) > 0 )

  /* cuvintul curent este mai mare decit cel pastrat in cuvmax */
```

```
        strcpy(cuvmax,cuvcrt);
    printf("cel mai mare cuvint este\n");
    printf("%s\n", cuvmax);
  }
```

8.14 Să se scrie un program care citeşte o succesiune de cuvinte, le sortează în ordine crescătoare şi apoi le afişează în ordinea respectivă.

Prin *cuvînt* înţelegem un şir de litere mici sau mari. La compararea cuvintelor nu se face deosebirea între literele mici şi mari. În felul acesta, cuvintele se vor afişa în ordine alfabetică. Vom presupune că lungimea unui cuvînt nu depăşeşte 30 de caractere şi că sînt cel mult 500 de cuvinte la intrare.

În prima parte se citesc cuvintele şi se păstrează în tabloul *tcuvinte*. Acesta este de tip *char*. Fiecare cuvînt se termină prin caracterul *NUL*.

La întîlnirea sfîrşitului de fişier se trece la sortarea cuvintelor păstrate în tabloul *tcuvinte*.

Sortarea cuvintelor se face folosind metoda bulelor (vezi exerciţiul 4.40).

Conform acestei metode, se parcurg elementele tabloului comparîndu-se de fiecare dată două elemente vecine. Dacă ele nu sînt în ordinea cerută (crescătoare), atunci ele se permută. Permutarea elementelor învecinate este simplă în cazul în care elementele respective sînt numere. În cazul de faţă elementele sînt cuvinte şi permutarea lor este mai complicată. De aceea,vom folosi pointeri spre începutul fiecărui cuvînt. Aceşti pointeri îi păstrăm ca elemente ale unui tablou pe care îl numim *tpointer*.

La terminarea citirii cuvintelor, elementele lui *tpointer* vor pointa spre începuturile cuvintelor păstrate în tabloul *tcuvinte*. Astfel, tpointer[0] pointează spre primul cuvînt citit şi păstrat în *tcuvinte* (are ca valoare adresa de început a zonei de memorie în care se păstrează primul caracter), tpointer[1] pointează spre al doilea cuvînt şi aşa mai departe.

Exemplu:

Presupunem că se tastează textul:

Programarea orientata spre obiecte este un stil modern
de programare.

După citirea textului respectiv, tabelele *tpointer* şi *tcuvinte* au următoarele conţinute:

tpointer[0] = adresa lui tcuvinte[0];

Elementele tcuvinte[0] - tcuvinte[11] conţin caracterele cuvîntului

Programarea, inclusiv caracterul *NUL* de la sfîrşitul cuvîntului.

tpointer[1] = adresa lui tcuvinte[12];

Elementele tcuvinte[12] - tcuvinte[21] conţin caracterele corespunzătoare cuvîntului *orientata*.

tpointer[2] = adresa lui tcuvinte[22];

Elementele tcuvinte[22] - tcuvinte[26] conţin caracterele corespunzătoare cuvîntului *spre* şi aşa mai departe.

Ultimul element al tabloului *tpointer* la care i s-a atribuit valoare este:

tpointer[9] = adresa lui tcuvinte[58]

Elementele tcuvinte[58] - tcuvinte[68] conţin caracterele corespunzătore cuvîntului *programare*.

Pentru a compara două cuvinte învecinate păstrate în tabloul *tcuvinte*, utilizăm pointeri spre ele, pointeri care sînt memoraţi în tabloul *tpointer*.

Astfel, pentru a compara cuvîntul al *k*-lea din tabloul *tcuvinte*, cu următorul lui, vom apela funcţia *stricmp* cu parametri tpointer[k] şi tpointer[k+1]:

```
stricmp(tpointer[k],tpointer[k+1])
```

Dacă la un astfel de apel funcţia *stricmp* returnează o valoare pozitivă, înseamnă că tpointer[k] pointează spre un cuvînt care este mai mare (urmează în ordine alfabetică) decît cuvîntul spre care pointează tpointer[k+1]. În acest caz ar urma să se permute cuvintele respective, deoarece nu sînt în ordinea cerută.

Existenţa pointerilor spre cuvintele respective face ca să nu mai fie necesară permutarea efectivă a cuvintelor, ci numai a valorilor pointerilor spre ele. Deci, în acest caz se vor permuta valorile pointerilor tpointer[k] şi tpointer[k+1]:

```
if(stricmp(tpointer[k],tpointer[k+1]) > 0)
{
 char *t;

 t = tpointer[k];
 tpointer[k] = tpointer[k+1];
 tpointer[k+1] = t;
}
```

Se observă utilizarea funcţiei *stricmp*, pentru a realiza comparaţii între cuvinte la care se ignoră diferenţa dintre literele mici şi mari.

PROGRAMUL BVIII14

```
#include <stdio.h>
#include <string.h>
#include <conio.h>

#define MAXLC 30
#define MAXNC 500

main() /* sorteaza un sir de cuvinte in ordine crescatoare
          (alfabetica) */
{
 static char tcuvinte[(MAXLC+1)*MAXNC];
 static char *tpointer[MAXNC];
 int c,i,j,k,ind;
 char *t;

 i = 0; /* indice in tpointer */
 j = 0; /* indice in tcuvinte */
 c = getchar();

/* citirea cuvintelor */
 while(c != EOF ) { /* 1 */

/* avans peste caracterele care nu sînt litere*/
    while( !( c >= 'A' && c <= 'Z' ||
              c >= 'a' && c <= 'z')) {

/* c nu contine o litera */
         c = getchar();
         if( c == EOF )
            break;
    }
    if( c == EOF )
         break;

/* c contine o litera; este litera de inceput a cuvintului curent; acest
   cuvint se pastreaza incepind cu elementul tcuvinte[j]; adresa de ince-
   put a cuvintului, adica adresa elementului tcuvinte[j], se pastreaza
   in tabloul tpointer; aceasta adresa este tcuvinte + j si se atribuie
  elementului tpointer[i] */
    tpointer[i++] = tcuvinte + j;
```

```
/* se citesc caracterele cuvintului curent si se pastreaza in tabloul
   tcuvinte incepind cu elementul tcuvinte[j] */
     while(c >= 'A' && c <= 'Z' ||
          c >= 'a' && c <= 'z' ) {
          tcuvinte[j++] = c;
          c = getchar();
     }
/* se pastreaza caracterul NUL dupa cel curent */
     tcuvinte[j++] = '\0';
  } /* sfirsit while 1 */

/* s-a terminat citirea cuvintelor */
/* s-au citit i cuvinte */

  if(i ) { /* 1 */

/* exista cel putin un cuvint citit */
/* se face sortare */
     ind = 1;
     while ( ind ) {
          ind = 0;
          for( k=0; k < i-1; k++)

/* se analizeaza ordinea cuvintelor vecine */
               if(stricmp(tpointer[k],
                    tpointer[k+1]) > 0 ) {
               /* permutarea pointerilor */
                    t = tpointer[k];
                    tpointer[k] = tpointer[k+1];
                    tpointer[k+1] = t;
                    ind = 1;
               } /* sfirsit if */
     } /* sfirsit while */

/* s-a terminat sortarea; se afiseaza cuvintele */
     for(j = 0; j < i; j++ ) {
          printf("%s\n", tpointer[j]);
          if((j+1)%23 == 0 ) {
               printf("actionati o tasta pentru a\
                    continua\n");
               getch();
          }
```

```
        } /* sfirsit for */
    } /* sfirsit if 1 */
}
```

8.7. Expresie lvalue

În paragraful 3.2.7. s-a definit expresia de atribuire prin următorul format:

(1) *v = expresie*

unde:

v - Se consideră că este o variabilă simplă, o variabilă cu indici (permite accesul sau modificarea valorii unui element de tablou) sau defineşte un element de structură.

Definiţia expresiei de atribuire poate fi completată în momentul de faţă adăugînd formatul:

(2) **ep = expresie*

unde:

ep - Este o expresie care defineşte un pointer nenul şi care nu este un pointer spre o zonă constantă. Astfel de expresii s-au utilizat deja în paragrafele precedente.

Exemplu:

```
int n;
double x;
void *p;
```

Expresiile de atribuire de forma:

n = 123, x = 3.14159

corespund formatului (1), indicat mai sus pentru expresiile de atribuire.

Fie:

p = &n;

atunci am văzut că atribuirea

*(int *)p = 123

realizează acelaşi lucru ca şi expresia:

n = 123.

Expresia:

(int *)p

defineşte un pointer spre o zonă care nu este constantă şi deci atribuirea de mai sus corespunde formatului (2) al expresiilor de atribuire. În mod analog, dacă:

p = &x;

atunci atribuirea:

*(double *)p = 3.14159

realizează acelaşi lucru ca şi expresia:

x = 3.14159

Expresiile utilizabile în *partea stîngă* a unei expresii de atribuire se numesc expresii *lvalue*.

Litera *l* provine de la cuvîntul englezesc *left*.

Noţiunea de expresie *lvalue* aparţine autorilor limbajului C (vezi [2]).

Ulterior s-a introdus şi noţiunea de expresie *rvalue*, care este o expresie care se poate utiliza în partea dreaptă a unei expresii de atribuire, dar nu şi în partea stîngă (vezi[8]).

Din cele de mai sus rezultă că o *expresie lvalue* poate fi:

- un nume de variabilă simplă;
- o variabilă cu indici;
- o construcţie ce permite accesul sau modificarea valorii unui element de structură;
- o construcţie de forma:

  *ep

unde:

ep - Este o expresie care defineşte un pointer nenul spre o dată care nu este o constantă.

Fie declaraţiile:

const int *a = 100;
int b;

În acest caz, *a* este un pointer spre o zonă constantă şi de aceea, expresia:

*a = 123

nu este corectă. În acest caz *a nu este o expresie *lvalue*. Ea este o expresie *rvalue*, deoarece se poate utiliza în partea dreaptă a expresiilor de atribuire.

Într-adevăr, expresia:

b = *a

este corectă şi atribuie variabilei *b* valoarea 100.

8.8. Alocarea dinamică a memoriei

Limbajul C permite utilizatorului să aloce date atît pe stivă (date automatice), cît şi în zone de memorie care nu aparţin stivei (date globale sau statice).

Alocarea datelor pe stivă se face la execuţie şi ea nu este permanentă. Astfel, dacă declaraţia:

tip nume;

se utilizează în corpul unei funcţii, atunci variabila *nume* se alocă pe stivă la fiecare apel al funcţiei respective. La revenirea din funcţie, stiva se "curăţă" (se reduce la starea avută înaintea apelului) şi prin aceasta variabila *nume* nu mai este alocată (devine nedefinită). O alocare de acest fel a memoriei se spune că este *dinamică.*

Pentru datele globale sau statice, memoria este alocată în fazele precedente execuţiei şi alocarea rămîne valabilă pînă la terminarea execuţiei programului. De aceea, pentru datele de acest fel se spune că alocarea este *statică* (nu este dinamică).

Limbajele C şi C++ oferă utilizatorului posibilitatea de a aloca dinamic memorie şi în alt mod decît cel indicat mai sus pentru datele automatice. Aceasta se realizează într-o zonă de memorie specială, distinctă de stivă. Această zonă de memorie se numeşte *memorie heap* (memorie grămadă, morman, de acumulare etc.) Ea poate fi gestionată prin funcţii standard în ambele limbaje.

Mai jos indicăm cîteva funcţii care pot fi utilizate la alocarea dinamică a datelor în memoria *heap*. Aceste funcţii sînt comune ambelor limbaje. Mai tîrziu o să vedem că în limbajul C++ există chiar operatori pentru alocări dinamice în memoria *heap*.

Funcţiile standard pentru gestiunea memoriei *help* au prototipurile în fişierul *alloc.h*.

Alocarea unei zone de memorie în memoria *heap* se poate realiza cu

ajutorul mai multor funcţii. O funcţie utilizată frecvent este funcţia *malloc*. Aceasta are prototipul:

void *malloc(unsigned *n*);

Funcţia alocă în memoria *heap* o zonă contiguă de *n* octeţi. Ea returnează adresa de început a zonei alocate. Această adresă reprezintă un pointer de tip *void* (void *). Prin intermediul acestuia se pot păstra date în zona de memorie alocată în acest fel. Pentru a păstra o dată de un *tip* dat într-o zonă de memorie alocată prin *malloc* este necesar să convertim adresa returnată de funcţie spre tipul datei respective.

Exemplu:

Se cere să se aloce în memoria *heap* o zonă de memorie pentru a păstra n valori de tip *int*. În acest scop declarăm un pointer spre tipul *int*:

```
int *p;
```

Apoi, apelăm funcţia *malloc* cu ajutorul expresiei de atribuire:

```
p = (int *)malloc(n*sizeof(int))
```

Se observă că valoarea returnată de funcţia *malloc* a fost convertită spre tipul *int* *, adică pointer spre *int*.

În continuare putem păstra şi utiliza date de tip *int*, folosind pointerul *p*. De exemplu, expresia:

```
*p = 123
```

păstrează întregul 123 în primii doi octeţi ai zonei alocate prin expresia de mai sus.

Expresia:

```
y = *p
```

atribuie lui *y* valoarea păstrată mai sus în memoria *heap*.

În general, expresia:

```
*(p+i) = k
```

atribuie valoarea lui *k* în zona de memorie de adresă definită prin expresia pointer:

```
p+i
```

Aceeaşi expresie se poate utiliza pentru a face acces la data respectivă.

Observaţii:

1. Funcţia *malloc* are ca parametru un întreg fără semn, adică acesta

trebuie să aparţină intervalului [0,65535].

2. În cazul în care în memoria *heap* nu se poate aloca o zonă de memorie contiguă de atîţia octeţi cît este valoarea parametrului de la apel, se va returna *pointerul nul*, adică valoarea *zero*.

De aceea, după apelul funcţiei *malloc* vom testa valoarea returnată pentru a ne asigura că aceasta nu este zero.

Zonele alocate prin funcţia *malloc* pot fi *eliberate*, pentru a putea fi eventual realocate, folosind funcţia standard *free*. Aceasta are prototipul:

void free(void ***p*);**

Prin apelul ei, se eliberează zona de memorie din memoria *heap*, spre care pointează *p*. Menţionăm că valoarea lui *p* trebuie să fie obţinută printr-un apel al unei funcţii standard de alocare, cum este de exemplu funcţia *malloc*.

Se recomandă ca această funcţie să fie apelată de îndată ce datele dintr-o zonă de memorie *heap* nu mai sînt necesare.

În felul acesta zona respectivă poate fi ulterior realocată. De asemenea, eliberarea sistematică a zonelor din memoria *heap* poate preveni fărîmiţarea excesivă a memoriei *heap*.

O altă funcţie standard utilă pentru a aloca zone de memorie în memoria *heap* este funcţia *calloc*. Ea are prototipul:

void *calloc(unsigned *nrelem*,**unsigned** *dimelem*);

Funcţia alocă o zonă de memorie de *nrelem* **dimelem* octeţi.

Ca şi funcţia *malloc*, funcţia *calloc* returnează adresa de început a zonei de memorie alocată, adresă care reprezintă un pointer spre *void*.

În cazul în care nu se pot aloca *nrelem* * *dimelem* octeţi, funcţia returnează valoarea zero.

Elementele zonei de memorie alocată prin *calloc* sînt iniţializate cu valoarea zero.

Zona de memorie alocată cu ajutorul funcţiei *calloc* se eliberează folosind funcţia *free* indicată mai sus.

Pentru programe relativ mici, pointerii se păstrează pe 16 biţi. Aceasta înseamnă că ei pot păstra adrese de pînă la 64k. Despre un astfel de pointer se spune că este de tip *near*.

Programele mai complexe, care necesită adrese de peste 64k, utilizează pointeri alocaţi pe 32 de biţi. Despre un astfel de pointer se spune că este de

tip *far*.

Biblioteca standard conţine funcţii pentru a aloca în memoria *heap* zone de memorie a căror adrese se reprezintă în memoria *heap* pe mai mult de 16 biţi. De asemenea, dimensiunea unei astfel de zone de memorii poate depăşi 64k. Astfel, pentru a aloca în memoria *heap* zone de memorie de adrese mai mari ca 64k, putem folosi funcţiile *farmalloc* şi *farcalloc*, funcţii analoge cu funcţiile *malloc* şi respectiv *calloc*, indicate mai sus.

Funcţia *farmalloc* are prototipul:

void far *farmalloc(unsigned long *n*);

Ea se utilizează la fel ca şi funcţia *malloc*. În acest caz funcţia returnează un pointer de tip *far* (se alocă pe 32 de biţi). De asemenea, la apelul funcţiei *farmalloc*, parametrul *n* poate depăşi 64k = 65536.

Zona alocată cu ajutorul funcţiei *farmalloc* se eliberează apelînd funcţia *farfree*, care este similară cu funcţia *free*. Ea are prototipul:

void farfree(void far **p*);

Exerciţii:

8.15 Să se scrie o funcţie care păstrează un şir de caractere într-o zonă de memorie alocată în memoria *heap*.

Funcţia returnează adresa de început a zonei în care se păstrează şirul de caractere sau zero (pointerul nul), în cazul în care nu se poate rezerva zona respectivă în memoria *heap*.

FUNCŢIA BVIII15

```
char *memsir(char *s)
/* pastreaza in memoria heap sirul de caractere spre care pointeaza s */
{
 char *p;

 if((p = (char *)malloc(strlen(s) + 1 )) != 0) {

/* p are ca valoare adresa de inceput a zonei de memorie rezervata in
   memoria heap si care are strlen(s) + 1 octeti */
      strcpy(p,s); /* copiaza sirul spre care pointeaza s, in zona
                      spre care pointeaza p */
      return p;
 }
 else
```

```
        return 0; /* pointerul nul; nu s-a putut aloca zona necesara
                     pentru a pastra sirul spre care pointeaza s */
}
```

Observaţii:

1. Funcţia de faţă returnează un pointer spre caractere, deci *tip* din antetul ei este:

   char *

2. Condiţia din paranteza lui *if* se poate scrie şi fără partea

   != 0

   adică

   if(p=(char *)malloc(strlen(s)+1))

   Apelul:

   strlen(s)

   determină numărul de octeţi necesari pentru a păstra caracterele proprii ale şirului spre care pointează *s*.

   Numărul returnat de funcţia *strlen* se măreşte cu 1 pentru a aloca un octet pentru caracterul *NUL* care trebuie să termine orice şir de caractere.

3. Apelul:

   strcpy(p,s);

   copiază caracterele şirului, inclusiv caracterul *NUL*, spre care pointează *s*, în zona a cărei adresă de început este valoarea lui *p*.

8.16 Să se scrie un program care citeşte o succesiune de cuvinte şi-l afişează pe cel mai mare.

Un astfel de program este descris în exerciţiul 8.13. În programul respectiv, cuvîntul maxim se păstrează în tabloul *cuvmax* alocat pe stivă. În programul de faţă se va păstra cuvîntul maxim în memoria *heap*.

PROGRAMUL BVIII16

```
#include <stdio.h>
#include <string.h>
#include <alloc.h>
```

```
#include "bviii15.cpp"

#define MAX 100

main() /* citeste o succesiune de cuvinte si-l afiseaza
          pe cel mai mare */
{
 char cuvcrt[MAX+1];
 char *cuvmax = 0;  /* pointeaza spre cuvintul maxim; initial
                       are ca valoare pointerul nul */

 while(scanf("%100s", cuvcrt) != EOF )
     if(cuvmax==0) /* prima citire */
        cuvmax = memsir(cuvcrt);
     else
        if(strcmp(cuvcrt,cuvmax) > 0 ) {

/* cuvintul citit este mai mare decit cel din memoria heap si spre care
   pointeaza cuvmax */

/* se elibereaza zona din memoria heap in care se patreaza cuvintul
   maxim intilnit la citirile anterioare */
          free(cuvmax);

/* se aloca zona in memoria heap si se pastreaza in ea cuvintul curent
   citit */
          cuvmax = memsir(cuvcrt);
        }
 printf("cel mai mare cuvint este\n");
 printf("%s\n", cuvmax);
 free(cuvmax);
}
```

Observaţie:

În acest program nu s-a testat imposibilitatea păstrării şirului maxim în memoria *heap* deoarece acesta are cel mult 101 caractere. Ori memoria *heap* are alocat un spaţiu mai mare.

8.17 Să se scrie o funcţie care sortează în ordine crescătoare *n* şiruri de caractere.

Funcţia are doi parametri:

tpointer - Tablou unidimensional de tip pointer spre *char*.
- Fiecare element al tabloului este un pointer spre unul din cele n şiruri de caractere.
- tpointer[i] este pointer spre al i+1 - lea şir de caractere.

n - Întreg de tip *int* care are ca valoare numărul şirurilor de caractere care se sortează.

Funcţia utilizează metoda bulelor. Pentru a ordona şirurile în ordine crescătoare se permută pointerii în locul şirurilor, ca în exerciţiul 8.14.

La compararea şirurilor de caractere se ignoră diferenţa dintre literele mici şi mari.

FUNCŢIA BVIII17

```
void ordsircresc(char *tpointer[], int n)
/* sorteaza in ordine crescatoare cele n siruri de caractere spre care
  pointeaza elementele
      tpointer[0],tpointer[1],...,tpointer[n-1] */
{
 int ind,i;
 char *t;

 ind = 1;
 while( ind ) {
     ind = 0;
     for(i=0; i< n-1; i++)
         if(stricmp(tpointer[i],tpointer[i+1]) > 0){
             t = tpointer[i];
             tpointer[i] = tpointer[i+1];
             tpointer[i+1] = t;
             ind = 1;
         }
 }
}
```

8.18 Să se scrie un program care citeşte o succesiune de cuvinte şi le afişează în ordine crescătoare. Se ignoră diferenţa dintre literele mici şi mari.

Prin cuvînt se înţelege o succesiune de caractere care nu sînt albe. Cuvintele sînt separate prin caractere albe. Succesiunea de cuvinte se termină cu sfîrşitul de fişier.

Cuvintele citite se păstrează în memoria *heap*. Programul utilizează un tablou de pointeri ale cărui elemente pointează fiecare spre cîte un cuvînt.

Se presupune că sînt cel mult 500 de cuvinte.

PROGRAMUL BVIII18

```
#include <stdio.h>
#include <string.h>
#include <conio.h>
#include <stdlib.h>
#include <alloc.h>

#include "bviii15.cpp"
#include "bviii17.cpp"

#define MAXCUV 500

main() /* citeste o succesiune de cuvinte si le afiseaza in
          ordine crescatoare */
{
 char *tp[MAXCUV];
 int i,j;
 char t[255];

/* se citesc cuvintele si se pastreaza in memoria heap */
 for(i=0; scanf("%s",t) == 1; i++)
     if((tp[i] = memsir(t)) == 0 ) {
         printf("memorie insuficienta\n");
         exit(1);
     }

/* se sorteaza cuvintele in ordine crescatoare */
 if( i )
     ordsircresc(tp,i);

/* listarea cuvintelor dupa sortare */
 for(j=0; j < i; j++) {
     printf("%s\n", tp[j] );
     if((j+1)%23 == 0 ) {
         printf("actionati o tasta pentru a\
               continua\n");
         getch();
     }
 } /* sfirsit for listare */
}
```

Observaţie:

Acest program are un avantaj faţă de cel din exerciţiul 8.14, deoarece în cazul de faţă se alocă pentru fiecare cuvînt o zonă de memorie de lungime egală cu numărul de caractere ale cuvîntului respectiv, plus 1 (pentru caracterul *NUL*). În programul din exerciţiul 8.14. se alocă o zonă fixă de memorie de (30+1)*500 octeţi, care este de obicei prea mare, deoarece cuvintele au în medie mai puţin de 30 de caractere.

În general, alocarea dinamică în memoria *heap* permite alocări optime, faţă de cele făcute în mod obişnuit, folosind tablouri automatice, globale sau statice.

8.9. Utilizarea tablourilor de pointeri la prelucrări de date de tip şir de caractere

Tablourile de pointeri pot fi utilizate pentru a prelucra succesiuni de şiruri de caractere. Utilizări de acest fel s-au văzut deja în exerciţiile din paragraful precedent (vezi 8.18. şi 8.14.).

Într-un program se întîlnesc adesea şiruri de caractere care se tratează într-un mod unitar. În acest caz este util să se definească un tablou de pointeri, fiecare element al său fiind un pointer spre un astfel de şir de caractere.

Astfel de tablouri pot fi utilizate pentru a păstra sau a avea acces la diferite denumiri, cuvinte cheie, mesaje de eroare etc.

În [2] se definesc astfel de tablouri pentru a păstra şi utiliza denumirile lunilor calendaristice şi cuvintele cheie ale limbajului C.

De exemplu, pentru a păstra denumirile lunilor calendaristice se poate utiliza un tablou de pointeri cu 13 elemente.

Numim *tpdl* acest tablou. Elementul tpdl[1] este pointer spre şirul de caractere "ianuarie", tpdl[2] este pointer spre şirul de caractere "februarie" etc.

Ultimul element, tpdl[12] este pointer spre şirul de caractere "decembrie".

Primul element al tabloului, nu pointează spre denumirea nici unei luni. De aceea, acest element poate avea ca valoare pointerul nul. O altă posibilitate este ca tpdl[0] să pointeze spre un text de eroare, de exemplu "luna ilegala".

Dacă *p* este un pointer spre un şir de caractere, atunci poate fi definit sau

declarat ca pointer spre un şir de caractere printr-o construcţie de forma:

char *p = sir;

Şirul de caractere *sir* se păstrează într-o zonă de memorie, iar lui *p* i se atribuie adresa de început a zonei respective.

În mod analog se pot iniţializa elementele tabloului *tpdl*, folosind o definiţie sau o declaraţie de forma:

```
char *tpdl[] = {
"luna ilegala",
     "ianuarie",
     "februarie",
     "martie",
     "aprilie",
     "mai",
     "iunie",
     "iulie",
     "august",
     "septembrie",
     "octombrie",
     "noiembrie",
     "decembrie"
 };
```

Prin această construcţie, elementul tpdl[i], pentru 1<=i<=12, este pointer spre şirul de caractere care denumeşte luna calendaristică a *i* - a din an. De exemplu, apelul:

printf("%s\n",tpdl[1]);

afişează textul

ianuarie

Exerciţii:

8.19 Să se scrie o funcţie care are ca parametru un întreg *n* şi returnează un pointer spre denumirea lunii calendaristice a n - a pentru 1<=n<= 12 sau spre textul "luna ilegala" pentru "n" în afara acestui interval.

Fie *denluna* numele acestei funcţii. Ea returnează un pointer spre *char*, deci returnează o dată de tip *char* * .

Deci antetul ei este:

char *denluna(int n)

FUNCŢIA BVIII19

```
char *denluna(int n)
/* returneaza pointerul spre denumirea lunii */
{
 static char *tpdl[] = {
     "luna ilegala",
     "ianuarie",
     "februarie",
     "martie",
     "aprilie",
     "mai",
     "iunie",
     "iulie",
     "august",
     "septembrie",
     "octombrie",
     "noiembrie",
     "decembrie"
 };

 return n <1 || n > 12 ? tpdl[0]: tpdl[n];
}
```

8.20 Să se scrie un program care citeşte o dată calendaristică scrisă sub forma:

zzllaaaa

o validează şi o afişează sub forma:

zz luna aaaa

unde:

zz	- Reprezintă ziua pe două cifre.
ll	- Reprezintă numărul lunii pe două cifre.
aaaa	- Reprezintă anul pe 4 cifre.
luna	- Reprezintă denumirea lunii calendaristice.

Data citită se validează utilizînd funcţia *v_calend* definită în exerciţiul 6.5.

PROGRAMUL BVIII20

```
#include <stdio.h>
```

```
#include <stdlib.h>

#include "bvi5.cpp"
#include "bviii19.cpp"

int nrzile[] = { 0,31,28,31,30,31,30,31,
                 31,30,31,30,31 };

main() /* - citeste o data calendaristica sub forma:
              zzllaaaa
           - o valideaza si o afiseaza sub forma:
              zz luna aaaa */
{
 int zz,ll,aaaa;

 if(scanf("%2d %2d %4d", &zz, &ll, &aaaa) != 3 ) {
     printf("nu s-a tastat un intreg de 8\
            cifre\n");
     exit(1);
 }
 if(v_calend(zz,ll,aaaa) == 0 ){
     printf("data calendaristica eronata\n");
     exit(1);
 }
 printf("%02d %s %4d\n", zz, denluna(ll),aaaa);
}
```

8.21 Să se scrie o funcţie care din denumirea lunii calendaristice determină numărul ei.

Această funcţie este inversa funcţiei *denluna* definită în exerciţiul 8.19.

Funcţia are ca parametru un pointer spre denumirea lunii calendaristice. Folosind acest pointer, denumirea lunii se caută cu ajutorul funcţiei *strcmp* pentru a stabili coincidenţa ei cu unul din şirurile de caractere spre care pointează elementele tabloului *tpdl* (vezi exerciţiul 9.19.). În cazul în care denumirea este eronată funcţia returnează valoarea zero.

FUNCŢIA BVIII21

```
int   nrluna(char   *denl)
/* returneaza numarul lunii din denumirea ei */
{
```

```
    static char *tpdl[] = {
        "",
        "ianuarie",
        "februarie",
        "martie",
        "aprilie",
        "mai",
        "iunie",
        "iulie",
        "august",
        "septembrie",
        "octombrie",
        "noiembrie",
        "decembrie"
    };
    int i;

    for( i=1; i <= 12; i++)
        if(strcmp(denl,tpdl[i]) == 0 )
            return i;
    return 0;
}
```

8.22 Să se scrie un program care citeşte o dată calendaristică tastată sub forma:

zz luna aaaa

şi o afişează sub forma:

zz/ll/aaaa

dacă este validă.

În aceste formate s-au utilizat notaţiile:

zz	- Ziua pe 1 - 2 cifre.
ll	- Luna pe 1 - 2 cifre.
aaaa	- Anul pe 4 cifre.
luna	- Denumirea lunii calendaristice.

PROGRAMUL BVIII22

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#include "bvi5.cpp"
#include "bviii21.cpp"

int nrzile[] = { 0,31,28,31,30,31,30,31,
                 31,30,31,30,31 };

main() /*- citeste o data calendaristica sub forma:
           zz luna aaaa
         - o valideaza si o afiseaza sub forma:
           zz/ll/aaaa */
{
 int zz,ll,aaaa;
 char dl[11]; /* pastreaza denumirea lunii calendaristice */

 if(scanf("%2d %10s %4d", &zz, dl,&aaaa) != 3 ) {
     printf("data calendaristica eronata\n");
     exit(1);
 }

/* determina numarul lunii calendaristice */
 if((ll = nrluna(dl)) == 0 ) {
     printf("luna eronata\n");
     exit(1);
 }
 if(v_calend(zz,ll,aaaa) == 0 ){
     printf("data calendaristica eronata\n");
     exit(1);
 }
 printf("%d/%d/%d\n", zz, ll, aaaa);
}
```

8.10. Tratarea parametrilor din linia de comandă

În linia de comandă folosită la apelul execuţiei unui program se pot utiliza diferiţi parametri (argumente).

În cazul utilizării mediului integrat de dezvoltare Turbo C, aceşti parametri se definesc utilizînd submeniul *Arguments* al meniului *Options*. Dacă se utilizează mediul integrat de dezvoltare *Borland C++*, atunci submeniul *Arguments* se selectează din meniul *Run*.

În toate cazurile argumentele sînt succesiuni de caractere separate prin spaţii. Ele se tastează în linia de comandă prin care se activează imaginea executabilă (fişierul cu extensia .exe) a programului.

Dacă se utilizează mediile integrate de dezvoltare Turbo C şi Borland C++, atunci se procedează astfel:

mediul Turbo C	**mediul Borland C++**
Se selectează meniul	Se selectează meniul
Option:	*Run*:
<Alt>-O	<Alt>-R
Se selectează submeniul:	Se selectează submeniul:
Arguments:	*Arguments*:
A	A

După selectarea submeniului *Arguments* se afişează o fereastră în care se pot tasta argumentele separate prin spaţiu. După ultimul argument se acţionează tasta Enter. Apoi se poate continua cu lansarea automată a fazelor: compilare, link-editare şi execuţie. Se tastează:

<Ctrl>-F9

Pentru a utiliza aceste argumente se vor folosi parametrii *argc* şi *argv* ai funcţiei principale. În acest caz funcţia principală are antetul

```
main(int argc,char *argv[])
```

Parametrul *argc* are ca valoare numărul argumentelor din linia de comandă mărit cu unu. Parametrul *argv* este un tablou de pointeri spre zonele în care s-au păstrat argumentele definite ca mai sus.

Argumentele se păstrează sub formă de şiruri de caractere. Pointerul argv[0] este întotdeauna pointerul spre calea fişierului cu imaginea executabilă a programului. În continuare argv[1] este pointerul spre primul argument tastat, argv[2] este pointerul spre al doilea argument şi aşa mai departe. În general argv[i] este pointer spre al *i*-lea argument.

Exemplu:

Presupunem că la lansarea programului s-au furnizat argumentele:

1 OCTOMBRIE 1993

În cazul acesta parametrii funcţiei principale au valorile:

argc= 4 - (3 argumente plus specificatorul fişierului cu imaginea executabilă a programului).

argv[0] - Pointer spre specificatorul fişierului cu imaginea executabilă a programului (cale, numele şi extensia .EXE).

argv[*1*] - Pointer spre şirul "1".

argv[*2*] - Pointer spre şirul "OCTOMBRIE".

argv[*3*] - Pointer spre şirul "1993".

Exerciţii:

8.23 Să se scrie un program care afişează parametri din linia de comandă.

PROGRAMUL BVIII23

```
#include <stdio.h>
main( int argc, char *argv[])
/* afiseaza parametri din linia de comanda */
{
 int i;

 for( i=0; i < argc; i++)
    printf("%s\n",argv[i]);
}
```

8.24 Să se scrie un program care afişează data calendaristică preluată din linia de comandă sub forma:

zz/ll/aaaa

unde:

zz - Ziua pe 1 - 2 cifre.

ll - Numărul lunii calendaristice pe 1 - 2 cifre.

aaaa - Anul pe 4 cifre.

Programul apelează funcţia *v_calend* pentru a valida data calendaristică preluată din linia de comandă. Întrucît ea se tastează sub forma indicată mai sus, acesta reprezintă un singur argument.

argv[1] este pointerul spre zona în care se păstrează data calendaristică respectivă.

PROGRAMUL BVIII24

```
#include <stdio.h>
#include <stdlib.h>

#include "bvi5.cpp"
```

```
int nrzile[] = { 0,31,28,31,30,31,30,31,
                 31,30,31,30,31 };

main(int argc, char *argv[])
/* valideaza si afiseaza data calendaristica preluata din
   linia de comanda */
{
 int zz,ll,aaaa;
 char a,b;

 if(argc < 2 ) {
    printf("lipseste data calendaristica in linia\
          de comanda\n");
    exit(1);
 }
 if(argc > 2 ) {
    printf("mai mult de un argument in linia de\
          comanda\n");
    exit(1);
 }
 if(sscanf(argv[1],"%2d %c %2d %c %4d",&zz,&a,
              &ll,&b, &aaaa) != 5) {
    printf("argumentul nu are formatul\
           zz/ll/aaaa\n");
    exit(1);
 }
 if( a != '/' ) {
    printf("dupa zi nu urmeaza caracterul /\n");
    exit(1);
 }
 if(b != '/' ) {
    printf("dupa luna nu urmeaza caracterul /\n");
    exit(1);
 }
 if(v_calend(zz,ll,aaaa) == 0 ) {
    printf("data calendaristica eronata\n");
    exit(1);
 }
 printf("%d/%d/%d\n", zz, ll,aaaa);
}
```

8.11. Pointeri spre funcţii

Numele unei funcţii este un pointer spre funcţia respectivă. De aceea, numele unei funcţii poate fi folosit ca parametru efectiv la apelul unei funcţii. În felul acesta, o funcţie poate transfera funcţiei apelate un pointer spre o funcţie. Funcţia apelată, la rîndul ei, poate apela funcţia care i-a fost transferată în acest fel.

Fie, de exemplu, *f* o funcţie care are ca parametru o altă funcţie. Apelul:

f(*g*);

unde *g* este numele unei funcţii, transferă funcţiei *f* pointerul spre funcţia *g*.

Înainte de apel, ambele funcţii (*f* şi *g*) trebuie să fie definite sau să li se indice prototipurile.

Fie *g* de prototip:

tipg g(*listag*);

unde prin *tipg* s-a notat tipul valorii returnate de *g* sau *void* în cazul în care funcţia *g* nu returnează o valoare, iar *listag* este lista cu tipurile parametrilor funcţiei *g* sau cuvîntul *void* dacă *g* nu are parametri.

Să definim prototipul funcţiei *f*. Aşa cum s-a indicat mai sus, ea are un parametru care este un pointer spre o funcţie care are un prototip de forma prototipului lui *g* (numai numele poate diferi).

O construcţie de forma:

tip **nume*

defineşte pe *nume* ca pointer spre tipul *tip*.

O construcţie de forma:

tip **nume*(*lista*)

defineşte pe *nume* ca funcţie care returnează un pointer spre *tip*.

Prezenţa parantezelor conduce la faptul că *nume* este o funcţie şi nu un pointer. Aceasta deoarece parantezele rotunde reprezintă operatori mai prioritari decît operatorul unar *. Pentru ca *nume* să fie pointer, este nevoie ca operatorul unar * să se aplice prioritar faţă de parantezele care indică prezenţa unei funcţii. Aceasta se obţine folosind o construcţie de forma:

tip (**nume*)(*lista*)

În cazul de faţă *nume* este un pointer spre o funcţie. Valoarea returnată de această funcţie are tipul *tip*, iar *lista* defineşte tipurile parametrilor

acestei funcţii.

În cazul funcţiei f de mai sus, parametrul ei formal se defineşte astfel:

*tipg (*nume_par_formal)(listag)*

Amintind faptul că într-un prototip se pot omite numele parametrilor formali, rezultă că prototipul funcţiei *f* poate fi scris astfel:

tipf f(tipg (*)(listag));

unde *tipf* este tipul valorii returnate de funcţia *f* sau *void* în cazul în care *f* nu returnează o valoare.

Antetul funcţiei *f* va fi:

*tipf f(tipg (*p)(listag))*

unde *listag* defineşte tipurile parametrilor funcţiilor care se transferă la apelurile lui *f* prin pointerii spre ele.

Exemple:

1. Funcţia *g* are prototipul:

   int *g*(**void**);

Funcţia *f* care are la apel ca parametru efectiv pe *g* şi returnează o valoare de tip double, are prototipul:

double *f*(**int** (*)(**void**));

2. Funcţia *h* are prototipul:

   double *h*(**int**,**float**);

Funcţia *f* care are la apel ca parametru efectiv pe *h* şi care returnează o valoare de tip int, are prototipul:

int *f*(**double** (*) (**int**,**float**));

Fie funcţia *f* care are antetul:

*tipf f(tip (*p)(lista))*

Deci, parametrul ei *p* este un pointer spre o funcţie care returnează o valoare de tipul *tip*, iar tipurile parametrilor acestei funcţii sînt definiţi de *lista*.

La un apel de forma:

f(g);

lui *p* i se atribuie ca valoare pointerul spre funcţia *g*. Se pune problema de a apela funcţia *g* în corpul funcţiei *f*, folosind parametrul formal *p*. În acest

caz, apelul obişnuit al lui *g*:

g(...);

sau

x=g(...);

se schimbă înlocuind numele *g* (necunoscut în momentul definirii funcţiei *f*) prin *p.

Deci în corpul funcţiei *f*, vom utiliza apeluri de forma:

(*p)(...);

sau

x=(*p)(...);

pentru a apela funcţia care s-a transferat prin parametru.

Exerciţii:

8.25 Să se scrie o funcţie care calculează şi returnează valoarea aproximativă a integralei definite dintr-o funcţie f(x), folosind metoda trapezului.

Să presupunem că integrala se calculează de la *a* la *b*. Atunci o valoare aproximativă a integralei se determină cu ajutorul formulei de mai jos:

In=h*((f(a)+f(b))/2+f(a+h)+f(a+2*h)+f(a+3*h)+...f(a+(n-1)*h))

unde h=(b-a)/n.

Funcţia de faţă calculează partea dreaptă a acestei realaţii. Ea are următorii parametri:

a	- Limita inferioară - număr de tip *double*.
b	- Limita superioară - număr de tip *double*.
n	- Numărul de subintervale în care s-a împărţit intervalul [a,b] - întreg de tip *int*.
p	- Pointerul spre funcţia f(x) din care se calculează integrala - aceasta are un parametru de tip *double* şi returnează o valoare de tip *double* (double f(double x)).

FUNCŢIA BVIII25

```
double itrapez(double a, double b,
               int n, double(*p)(double))
/* - calculeaza prin metoda trapezului, integrala definita de la a la b din
     functia spre care pointeaza p;
   - n - este numarul subintervalelor in care s-a impartit  intervalul [a,b]*/
```

```
{
 double h,s;
 int i;

 h = (b - a) / n;
 s = 0.0;
 for(i=1; i < n; i++)
     s += (*p)(a+i*h);
 s += ((*p) (a) + (*p) (b)) / 2;
 s *= h;
 return s;
}
```

8.26 Să se scrie un program care calculează integrala definită din funcţia:

f(x) = sin(x*x)

de la 0 la 1, folosind metoda trapezului. Se cere o eroare mai mică decît 1e-8.

Pentru a obţine precizia dorită vom proceda ca mai jos:

1. Se alege valoarea iniţială pentru *n*, de exemplu 10.
2. Se calculează In.
3. Se calculează I2n (n este dublat).
4. Dacă abs(I2n-In) < 1e-8, atunci I2n este rezultatul şi procesul de calcul se întrerupe.

   Altfel se dublează *n*, se pune In=I2n şi se continuă cu pasul 3 de mai sus.

PROGRAMUL BVIII26

```
#include <stdio.h>
#include <math.h>

#include "bviii25.cpp"

#define A 0.0
#define B 1.0
#define N 10
#define EPS 1e-8

double sinxp(double); /* prototip */
```

```
main() /* calculeaza integrala definita din functia
              sinxp=sin(x*x)
        in intervalul [A,B] cu o eroare mai mica decit EPS */
{
 int n = N;
 double in,i2n,vabs;

 in = itrapez(A,B,n,sinxp);
 do {
    n = 2*n;
    i2n = itrapez(A,B,n,sinxp);
    if((vabs = in - i2n ) < 0 )
       vabs = -vabs;
    in = i2n;
 } while(vabs >= EPS );
 printf("integrala din sin(x*x)=%.10g\n",i2n);
 printf("numarul de subintervale n=%d\n", n);
} /* sfirsit main */

double sinxp(double x)
{
 return sin(x*x);
}
```

Observaţie:

Valoarea aproximativă a integralei din sin(x*x) în intervalul [0,1] este 0.3102683026. Ea se obţine pentru n=10240 în aproximativ 40 de secunde, la un calculator fără coprocesor.

8.27 Să se scrie o funcţie care determină rădăcina ecuaţiei:

(1) f(x)=0

din intervalul [a,b], cu o eroare mai mică decît EPS=1e-8, ştiind că ecuaţia are o singură rădăcină în intervalul respectiv şi că f(x) este o funcţie continuă pe acelaşi interval.

Din condiţiile enunţate mai sus rezultă că:

f(a)*f(b) <= 0

sau

f(a)*f(b) < 0

dacă nici una din limitele intervalului [a,b] nu sînt rădăcini ale ecuaţiei (1).

O metodă simplă de calcul a rădăcinii ecuaţiei (1) este prin metoda înjumătăţirii. Ea constă în următoarele:

1. c = (a+b)/2.

2. Dacă f(c)=0, atunci c este soluţia căutată şi se întrerupe procesul de calcul.

3. Dacă f(a)*f(c) < 0, atunci se pune b=c, altfel a=c.

4. Dacă b-a < EPS, atunci procesul de calcul se întrerupe şi rădăcina se ia media dintre a şi b.

   Altfel algoritmul se reia de la pasul 1.

FUNCŢIA BVIII27

```
#define EPS 1e-8

double radec(double a, double b,
             double (*p)(double))
/* - calculeaza si returneaza solutia ecuatiei f(x) = 0 din intervalul [a,b];
   - ecuatia are o singura solutie in intervalul [a,b];
   - p - este pointerul spre functia f(x) de prototip:
         double f(double). */
{
 double c,d;

 if((*p)(a) == 0 )
    return a; /* a este radacina */
 if((*p)(b) == 0 )
    return b; /* b este radacina */
 if((*p)(a) * (*p)(b) > 0 ) {
    printf("ecuatia sau nu are radacina in\
          [a,b]\n");
    printf("sau are radacini multiple in\
          [a,b]\n");
    exit(1);
 }
 do {
    c = (a + b )/2;
    if((d = (*p)(c)) == 0 )
       return c; /* c este radacina */
    if((*p)(a)*d < 0 )
       b = c;
```

```
    else
        a = c;
 } while( b-a >= EPS );
 return (a+b)/2;
}
```

8.28 Să se scrie un program care calculează şi afişează rădăcina ecuaţiei:

x-sin(x+1)=0

din intervalul [0.5,1], cu o eroare mai mică decît 1e-8.

PROGRAMUL BVIII28

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "bviii27.cpp"

#define A 0.5
#define B 1.0

double f(double x) /* returneaza valoarea functiei x - sin(x+1) */
{
 return x - sin( x+1);
}

main () /* calculeaza si afiseaza radacina ecuatiei:
                x - sin(x+1 ) = 0
           din intervalul [0.5,1], cu o eroare mai mica decit 1.e-8 */
{
 printf("radacina ecuatiei: \n");
 printf("x - sin(x+1) = 0 este: %.10g\n",
       radec(A,B,f));
}
```

9. RECURSIVITATE

Spunem despre o funcţie că este *recursivă* dacă ea se autoapelează. Ea se poate autoapela fie *direct*, fie *indirect* prin apelul altor funcţii.

În limbajele C şi C++ se pot defini funcţii recursive fără a fi nevoie de a specifica acest lucru.

La apelurile recursive ale unei funcţii aceasta este reapelată înainte de a se reveni din ea. La fiecare reapel a funcţiei, *parametrii* şi *variabilele automatice* ale ei se alocă pe stivă într-o zonă independentă. De aceea, aceste date au valori distincte la fiecare reapelare. Nu acelaşi lucru se întîmplă cu *variabilele statice*. Ele ocupă tot timpul aceeaşi zonă de memorie şi deci ele îşi păstrează valoarea la un reapel.

Orice apel al unei funcţii conduce la o revenire din funcţia respectivă la punctul următor celui din care s-a făcut apelul.

Amintim că la revenirea dintr-o funcţie se procedează la curăţirea stivei, adică stiva se reface la starea ei dinaintea apelului. De aceea, orice reapel al unui apel recursiv va conduce şi el la curăţirea stivei, la o revenire din funcţie şi deci parametrii şi variabilele locale vor reveni la valorile lor dinaintea reapelului respectiv. Numai variabilele statice rămîn neafectate la o astfel de revenire.

Funcţiile recursive se utilizează la definirea proceselor recursive de calcul.

Un proces de calcul se spune că este *recursiv*, dacă el are o parte care se defineşte prin el însuşi. Un exemplu simplu de astfel de funcţie este funcţia de calcul al *factorialului*. Într- adevăr funcţia *fact(n)* de calcul al factorialului se poate defini astfel:

fact(n)=1, dacă n=0;

altfel

fact(n)=n*fact(n-1).

Alternativa:

fact(n)=n*fact(n-1)

defineşte funcţia *fact(n)* prin ea însăşi.

Un proces recursiv trebuie totdeauna să conţină o parte care nu se defineşte prin el însuşi. În exemplul de mai sus, alternativa:

fact(n)=1, dacă n=0

este partea care este definită direct.

Definiţia de mai sus se transcrie imediat în limbajul C:

```
double fact(int n) /* calculează n! */
{
 if(n==0)
    return 1.0;
 else
    return n*fact(n-1);
}
```

Să urmărim execuţia acestei funcţii pentru n=3.

La apelul din funcţia principală se construieşte o zonă pe stivă în care se păstrează adresa de revenire în funcţia principală, după apel, precum şi valoarea lui n, ca în figura de mai jos.

Stiva programului

rev	adr1	Adresa de revenire în funcţia principală care a făcut apelul
n	3	

a

Deoarece n > 0, se execută alternativa *else*. Aceasta conţine expresia:

(1) n*fact(n-1)

care autoapelează (direct) funcţia *fact*. Reapelarea funcţiei se realizează cu valoarea n-1 = 3-1 = 2. Prin aceasta se construieşte o zonă nouă pe stivă, care conţine revenirea la evaluarea expresiei de mai sus, precum şi valoarea nouă a parametrului n, adică valoarea 2. Se obţine starea din figura de mai jos.

Stiva programului

rev	adr2	Adresa de revenire la evaluarea expresiei (1)
n	2	
rev	adr1	
n	3	

b

La noua reapelare a funcţiei fact, n=2 > 0, deci din nou se execută alternativa *else*.

Evaluarea expresiei (1) conduce la o reapelare cu valoarea n-1= 2-1= 1. Se obţine configuraţia:

Stiva programului

rev	adr2	Adresa de revenire la evaluarea expresiei (1)
n	1	
rev	adr2	
n	2	
rev	adr1	
n	3	

c

Din nou n = 1 >0 şi deci se face o nouă reapelare şi se obţine configuraţia:

Stiva programului

rev	adr2	Adresa de revenire la evaluarea expresiei (1)
n	0	
rev	adr2	
n	1	
rev	adr2	
n	2	
rev	adr1	
n	3	

d

În acest moment n=0, deci se revine din funcţie cu valoarea 1. Se curăţă stiva şi se revine la configuraţia din figura c. Adresa de revenire permite continuarea evaluării expresiei (1). În acest moment *n* are valoarea 1 şi cum s-a revenit tot cu valoarea 1, se realizează produsul

1*1 = 1

După evaluarea expresiei (1) se realizează o nouă revenire tot cu valoarea 1. După curăţirea stivei se ajunge la configuraţia din figura b. În acest moment *n* are valoarea 2. Evaluînd expresia (1), se realizează produsul

2*1 = 2.

Apoi se revine din nou din funcţie cu valoarea 2. După curăţirea stivei se ajunge la configuraţia din figura a.

În acest moment *n* are valoarea 3. Se evaluează expresia (1) şi se obţine:

3*2 = 6

Se revine din funcţie cu valoarea 6 (3!) şi conform adresei de revenire, se revine în funcţia principală din care s-a apelat funcţia *fact*.

În general, o funcţie recursivă se poate realiza şi nerecursiv, adică fără să se autoapeleze. De obicei, recursivitatea nu conduce la economie de memorie şi nici la execuţia mai rapidă a programelor. Ea permite însă o descriere mai compactă şi mai clară a funcţiilor care exprimă procese de calcul recursive. Cu toate acestea, există cazuri cînd funcţiile recursive, deşi au exprimări clare, ele nu sînt recomandate deoarece implică consum mare de timp şi memorie, în comparaţie cu varianta nerecursivă a aceluiaşi proces de calcul. De exemplu, algoritmul de generare a permutărilor de *n* obiecte poate fi descris atît recursiv, cît şi nerecursiv. O variantă recursivă simplă generează permutările de *n* obiecte inserînd un obiect nou (*n*) în toate poziţiile posibile în permutările de n-1 obiecte. Acest algoritm, deşi este mai simplu decît varianta nerecursivă, el nu este practic, deoarece generarea permutărilor de *n* obiecte presupune existenţa deja în memorie a permutărilor de n-1 obiecte, ori n! creşte foarte rapid, ceea ce conduce la un necesar mare de memorie chiar şi pentru *n* mic. Se pot indica şi alte funcţii recursive ineficiente faţă de variantele lor nerecursive. De exemplu, în multe lucrări se prezintă funcţia recursivă pentru calculul *numerelor lui Fibonacci* şi se arată ineficienţa ei faţă de varianta nerecursivă.

De aceea, rezolvările cu ajutorul funcţiilor recursive urmează să se adopte numai după un studiu atent al implicaţiilor pe care le au acestea în privinţa necesarului de memorie şi al timpului de execuţie.

Există cîteva tipuri de probleme care, de obicei, se rezolvă cu ajutorul funcţiilor recursive. Astfel, se recomandă să se utilizeze funcţii recursive

pentru probleme a căror metode de rezolvare se pot defini recursiv.

În această clasă se includ metode de divizare, metode de căutare cu revenire (*backtracking*) etc.

În toate cazurile trebuie însă să se aibă în vedere necesarul de memorie şi a timpului de execuţie în comparaţie cu variantele nerecursive.

Exerciţii:

9.1 Să se scrie un program care citeşte pe *n* de tip *int*, calculează şi afişează *n!* folosind funcţia recursivă *fact* definită mai sus.

PROGRAMUL BIX1

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 170

double fact(int n) /* calculeaza pe n! */
{
 if(n ==0 )
       return 1.0;
 else
       return n * fact(n-1);
}

main () /* citeste pe n, calculeaza si afiseaza pe n! */
{
 int n;
 char t[255];

 for( ; ; ) {
    printf("valoarea lui n: ");
    if(gets(t) == 0) {
       printf("s-a tastat EOF\n");
       exit(1);
    }
    if(sscanf(t,"%d",&n) == 1 &&     n >= 0 && n <=
    MAX )
       break;
    printf("se cere un intreg in intervalul\
          [0,%d]\n", MAX);
 }
```

```
  printf("n= %d\tn!= %g\n", n, fact(n));
}
```

9.2 Să se scrie o funcţie recursivă care calculează numărul aranjamentelor de *n* obiecte luate cîte *k* (n >= k > 0).

Dacă notăm cu A(n,k) numărul aranjamentelor de *n* obiecte luate cîte *k*, atunci:

$$A(n,k) = n(n-1)(n-2)...(n-k+1)=n(n-1)(n-2)...((n-1)-(k-1)+1)=$$
$$= nA(n-1,k-1);$$
$$A(n,1) = n.$$

Relaţia de mai sus, pentru calculul numărului aranjamentelor, este recursivă. Ea se transcrie imediat în limbajul C folosind o funcţie recursivă.

FUNCŢIA BIX2

```
double aranjamente(int n, int k)
/* returneaza numarul aranjamentelor de n obiecte luate cite k
  (n>=k >0) */
{
 if( k == 1 )
     return (double) n;
 else
     return n*aranjamente(n-1,k-1);
}
```

9.3 Să se scrie un program care citeşte doi întregi *n* şi *k*

(1 <= n<=170,1<= k <= n),

calculează şi afişează numărul aranjamentelor de *n* obiecte luate cîte *k*.

Valorile lui *n* şi *k* se citesc folosind funcţia *pcit_int_lim* definită în exerciţiul BVIII3.CPP. Această funcţie, la rîndul ei, apelează funcţia *pcit_int* definită în exerciţiul BVIII2.CPP.

PROGRAMUL BIX3

```
#include <stdio.h>
#include <stdlib.h>

#include "bviii2.cpp"
#include "bviii3.cpp"
#include "bix2.cpp"
```

```
#define MAX 170

main() /* - citeste intregii n si k (1 <= n <= 170; 1 <= k <= n),
          - calculeaza si afiseaza numarul aranjamentelor de
            n obiecte luate cite k */
{
 int n,k;
 char er[]="S-a tastat EOF\n";

 if(pcit_int_lim("n= ",1,170,&n) == 0 ) {
    printf(er);
    exit(1);
 }
 if(pcit_int_lim("k= ",1,n,&k) == 0 ) {
    printf(er);
    exit(1);
 }
 printf("n=%d\tk=%d\tA(n,k)=%g\n",n,k,
          aranjamente(n,k));
}
```

9.4 Să se scrie o funcţie care are ca parametri doi întregi *m* şi *n* de tip *long* şi care calculează şi returnează cel mai mare divizor comun al lor.

Notăm cu:

(m,n)

cel mai mare divizor comun al numerelor *m* şi *n*. Calculul celui mai mare divizor comun a două numere se poate realiza recursiv astfel:

(m,n) = m	dacă n = 0;
(m,n) = n	dacă m = 0;
(m,n) = (n,m%n)	dacă atît m, cît şi n sînt diferiţi de zero şi m > n (prin m%n s-a notat restul împărţirii lui *m* la *n*).

FUNCŢIA BIX4

```
long cmmdc(long m, long n)
/* calculeaza si returneaza pe cel mai mare divizor comun al
  numerelor m si n */
{
 if( m == 0 )
    return n;
```

```
  else
      if( n == 0 )
          return m;
      else
          if( m > n )
              return cmmdc( n, m%n);
          else
              return cmmdc( m, n%m);
}
```

9.5 Să se se scrie un program care citeşte doi întregi pozitivi *m* şi *n*, de tip *long*, calculează şi afişează pe cel mai mare divizor comun al lor.

PROGRAMUL BIX5

```
#include <stdio.h>
#include <stdlib.h>

#include "bix4.cpp"

main() /* citeste pe m si n de tip long, calculeaza si afiseaza (m,n) */
{
  long m,n;

  printf("valorile lui m si n: ");
  if(scanf("%ld %ld", &m, &n) != 2 || m <= 0
                    || n <= 0 ) {
      printf("nu s-au tastat doi intregi\
             pozitivi\n");
      exit(1);
  }
  printf("m=%ld\tn=%ld\t(m,n)=%ld\n",m,n,cmmdc(m,n))
}
```

9.6 Să se scrie un program pentru rezolvarea problemei *turnurilor din Hanoi*. Această problemă se enunţă ca mai jos.

Se consideră trei tije A, B, C şi *n* discuri de diametre diferite. Tijele sînt fixate vertical pe o placă. Discurile au fiecare cîte un orificiu în centru şi ele pot fi aranjate pe fiecare din cele trei tije. Iniţial toate discurile sînt puse pe tija A şi ordonate în aşa fel încît orice disc este aşezat peste un disc de diametru mai mare decît al lui. Dacă numerotăm discurile în ordinea crescătoare a diametrelor lor, adică discul 1 are diametrul cel mai mic, apoi discul 2 are diametrul următor şi aşa mai departe, discul n avînd diametrul

cel mai mare,atunci discul 1 se află în vîrf, sub el discul 2 etc., iar la bază se află discul *n*.

Se cere să se mute discurile de pe tija A pe tija B, folosind tija C ca tijă de manevră. La fiecare mutare se mută un singur disc şi anume, unul aflat în vîrful discurilor de pe o tijă se pune în vîrful discurilor de pe o altă tijă sau pe placă dacă tija respectivă nu conţine discuri. De asemenea, nu se poate muta un disc de diametru mai mare peste unul de un diametru mai mic. Deci, la o mutare, peste discul *k* se poate aşeza numai unul din discurile 1,2,...,k-1. În final discurile trebuie să fie aşezate pe tija B în aceeaşi ordine în care s-au aflat pe tija A, adică în vîrf să fie discul 1, sub el discul 2 şi aşa mai departe, discul *n* aflîndu-se la bază.

Problema turnurilor din Hanoi pentru *n* discuri se rezolvă imediat dacă ştim să o rezolvăm pentru n-1 discuri. Într-adevăr, în ipoteza că ştim să rezolvăm problema pentru n-1 discuri, procedăm astfel:

1. Se deplasează discurile 1,2,...,n-1 de pe tija A pe tija C.
2. Discul *n* se mută de pe tija A pe tija B.
3. Cele n-1 discuri de pe tija C se deplasează pe tija B.

În felul acesta, pe tija B se află toate cele *n* discuri şi în ordinea cerută, adică 1,2,...,n.

Deci rezolvarea problemei turnurilor din Hanoi cu *n* discuri s-a redus la rezolvarea aceleeaşi probleme, dar cu n-1 discuri. În mod analog, problema turnurilor din Hanoi cu n-1 discuri se poate rezolva dacă ştim să rezolvăm problema respectivă pentru n-2 discuri. În felul acesta, din aproape în aproape, ajungem la concluzia că problema turnurilor din Hanoi pentru *n* discuri se poate rezolva, dacă ştim să o rezolvăm pentru n=1. Ori în acest caz problema este imediată: se mută discul de pe tija pe care se află (tija sursă), pe tija pe care dorim să se afle discul (tija destinaţie).

Cele trei tije se află în una din următoarele stări posibile:

tijă sursă - Conţine discurile care trebuiesc transferate pe altă tijă.

tijă destinaţie - Tijă pe care trebuie să se afle discurile transferate de pe tija sursă.

tijă de manevră - Tijă care se foloseşte pentru a stoca temporar discuri.

Rolul tijelor se schimbă în diferite etape ale procesului de deplasare a discurilor şi de aceea, rolul fiecărei tije se defineşte cu ajutorul parametrilor.

Mai jos definim o funcţie recursivă pe care o numim *hanoi*. Ea are 4 parametri:

n - Numărul discurilor.

a - Tija sursă.
b - Tija destinaţie.
c - Tija de manevră.

Apelul:

hanoi(n,'A','B','C');

înseamnă că se rezolvă problema pentru n discuri. Acestea se află pe tija A, tija B este tija destinaţie şi tija C este tija de manevră.

Funcţia *hanoi* afişează fiecare mutare, indicînd numărul discului, tija de pe care se mută discul respectiv, precum şi tija pe care se mută el.

După afişarea unei mutări, se apelează funcţia *getch* pentru a bloca execuţia programului. În felul acesta se poate analiza mutarea afişată de calculator.

Menţionăm că problema de faţă poate fi rezolvată şi nerecursiv. Varianta nerecursivă este mai complexă decît cea recursivă.

PROGRAMUL BIX6

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

void hanoi(int n, int a, int b,int c )
/* functie recursiva pentru rezolvarea problemei turnurilor din Hanoi,
        n - numarul discurilor;
        a - tija sursa;
        b - tija destinatie;
        c - tija de manevra. */
{
 if(n == 1 ) { /* n = 1 */
    printf("se muta discul 1 de pe tija: %c pe\
          tija: %c\n",a,b);
    getch();
    return;
 }

/* n > 1;
  se rezolva problema pentru n-1 discuri;
  a - tija sursa; c - tija destinatie; b - tija de manevra. */
 hanoi(n-1, a, c, b);
```

```
/* discul n de pe tija a se muta pe tija b */
 printf("discul: %d de pe tija: %c se muta pe\
       tija: %c\n",n,a,b);
 getch();

/* se rezolva problema pentru n-1 discuri;
  c - tija sursa; b - tija destinatie; a - tija de manevra. */
 hanoi(n-1, c, b, a);
}
/* sfirsit hanoi */

main () /* citeste pe n si rezolva problema turnurilor din Hanoi
          pentru n discuri */
{
 int n;
 char t[255];

 for( ; ; ) {
    printf("numarul discurilor= ");
    if(gets(t) == 0 ) {
       printf("s-a tastat EOF\n");
       exit(1);
    }
    if (sscanf(t,"%d",&n) == 1 && n > 0 )
       break;
       printf("nu s-a tastat un intreg\
             pozitiv\n");
 }
 hanoi(n, 'A', 'B', 'C');
}
```

10. STRUCTURI ŞI TIPURI DEFINITE DE UTILIZATOR

Limbajele de programare oferă utilizatorului facilităţi de prelucrare atît a datelor *singulare*(izolate), cît şi a celor *grupate*.

Datele se grupează pentru a forma mulţimi de elemente care să poată fi prelucrate atît element cu element, cît şi global. De obicei, aceste grupe sînt mulţimi ordonate de date, adică datele unei astfel de grupe satisfac anumite relaţii.

Un mod simplu de grupare a datelor ne conduce la noţiunea de *tablou*.

Tabloul este o mulţime ordonată de date de un *acelaşi tip*, relaţia de ordine între elementele sale fiind definită cu ajutorul indicilor. *Numărul* indicilor defineşte *dimensiunea* tabloului. Tipul comun elementelor tabloului este şi *tipul tabloului*.

Adesea este util să grupăm date într-un alt mod decît cel utilizat în cazul tablourilor. Astfel, se pot grupa date care nu neapărat sînt de acelaşi tip. Gruparea se face cu scopul de a prelucra datele respective nu numai separat, individual ci şi global. Grupa şi în acest caz este o mulţime ordonată. Referirea la elementele ei nu se mai face însă folosind indici, ci construcţii de felul numelui. Componentele unei grupe pot fi ele însele grupe de alte date şi aşa mai departe. În felul acesta, datele sînt grupate pe *nivele*. Grupa care nu este componentă a unei alte grupe se spune că se află la nivelul *cel mai înalt*. Datele care nu mai sînt grupe de alte date se numesc *date elementare*. Ele se află la nivelele cele *mai inferioare*. În felul acesta datele sînt grupate potivit unei ierarhii. Datele grupate, conform unei ierarhii, se numesc *structuri*.

Un exemplu simplu de structură este *data calendaristică*. Ea este o grupă de 3 date elementare: *zi, lună şi an*.

Componentele *zi şi an* sînt date de tip întreg. Componenta *lună* poate fi un şir de caractere, care să reprezinte luna prin denumirea ei: ianuarie, februarie etc. De aceea, componenta *lună* este un tablou de tip caracter care permite memorarea denumirii unei luni calendaristice.

Observaţie:

Datele elementare ale unei structuri pot fi izolate sau tablouri.

Dacă în cazul tablourilor, tipul lui este tipul comun elementelor sale, în cazul structurilor fiecare structură reprezintă un *tip nou* de date. Un astfel

de tip se introduce printr-o *declarație de structură* și se spune că este un *tip definit de utilizator*.

În capitolul de față vom trece în revistă aspectele de bază cu privire la definirea și folosirea structurilor și a construcțiilor înrudite cu ele (reuniuni, cîmpuri de biți etc.).

10.1. Declarația de structură

Pentru a declara o structură se pot folosi mai multe formate. Un format utilizat frecvent este:

struct *nume* {
(1) *lista_de_declaratii*
} *nume1,nume2,...,numen*;

unde:

nume,nume1, nume2,...,numen - Sînt *nume* care pot și lipsi, dar nu toate deodată.

Dacă *nume* este absent, atunci cel puțin *nume1* trebuie să fie prezent.

Dacă lista *nume1,nume2,...,numen* este absentă, atunci trebuie ca *nume* să fie prezent.

Dacă *nume* este prezent, atunci el definește un *tip nou*, introdus prin declarația de structură respectivă. El poate fi utilizat în continuare pentru a defini date de acest tip, în mod asemănător cum definim date de tipuri predefinite.

În declarația de mai sus, *nume1,nume2,...,numen* sînt structuri de tipul *nume*.

În declarația de mai sus *numei (i=1,2,...,n)* se poate înlocui cu:

numei [lim1] [lim2]...[limk]

și atunci *numei* este un tablou k-dimensional de elemente de tip *nume*.

Prin *lista_de_declaratii* înțelegem una sau mai multe declarații prin care se declară componentele structurii de tip *nume*.

O structură de tip *nume* poate fi declarată și ulterior, folosind formatul:

(2) **struct** *nume nume_de_structura*;

unde:

nume_de_structura - Este un *nume*.

Această declaraţie seamănă cu o declaraţie obişnuită în care se folosesc tipuri predefinite:

tip nume_data_izolata;

Deci, *tip* dintr-o declaraţie obişnuită se înlocuieşte cu *struct nume*, unde *nume* este definit în prealabil printr-o declaraţie de structură de forma (1), în care *nume1,nume2,...,numen* pot şi lipsi.

Exemple:

1. Declarăm *data_nasterii* şi *data_angajarii* ca structuri de tip *data_calendaristica*, care este compusă din cele trei date amintite mai sus: *zi,luna şi an*:

```
struct data_calendaristica {
   int zi;
   char luna[11];
   int an;
} data_nasterii, data_angajarii;
```

Declaraţia de faţă este de forma (1).

În cazul în care nu dorim să introducem tipul *data_calendaristica*, se poate utiliza declaraţia:

```
struct {
   int zi;
   char luna[11];
   int an;
} data_nasterii, data_angajarii;
```

În sfîrşit, este posibil să definim tipul utilizator *data_calendaristica* şi apoi să declarăm datele *data_nasterii şi data_angajarii* folosind formatul (2).

```
struct data_calendaristica {
   int zi;
   char luna[11];
   int an;
} ;

struct data_calendaristica data_nasterii,
    data_angajarii;
```

Prima declaraţie introduce tipul utilizator *data_calendaristica.*

Declaraţia a doua defineşte datele *data_nasterii şi data_angajarii* ca fiind structuri de tipul *data_calendaristica.*

2. Fie tipul *data_calendaristica* definit mai sus. Declaraţia:

```
struct data_calendaristica d[100];
```

defineşte pe *d* ca un tablou de 100 de elemente, fiecare element fiind de tipul *data_calendaristica*.

3. Considerăm un model simplificat de date personale:

- nume_prenume;
- adresa;
- localitatea_nasterii;
- cod;
- data_nasterii;
- data_angajarii;
- sex;
- stare_civila;
- numar_copii.

Prin declaraţia de mai jos introducem tipul *date_pers:*

```
struct date_pers {
   char nume_prenume[100];
   char adresa[100];
   char localitatea_nasterii[100];
   long cod;
   struct data_calendaristica
      data_nasterii,
      data_angajarii;
   char sex;
   char starea_civila[15];
   int numar_copii;
};
```

La definirea tipului *date_pers* s-a utilizat tipul *data_calendaristica.* Acest lucru este posibil dacă tipul *data_calendaristica* este în prealabil definit.

Fie declaraţia:

```
struct date_pers unangajat,nangajati[1000];
```

Prin această declaraţie se declară structura *unangajat* şi tabloul de structuri *nangajati.* Ambele au tipul *date_pers.*

4. În acest exemplu se introduce tipul *complex:*

```
struct complex {
    double real;
    double imag;
};
```

Datele declarate cu ajutorul declaraţiei:

```
struct complex z,tz[100];
```

sînt date de tip *complex*. Atît partea reală, cît şi partea imaginară sînt date de tip *double*.

5. Fie declaraţia:

```
struct dmatp2 {
    double matrice[2][2];
} ;
struct dmatp2 a,b;
```

Variabilele *a şi b* sînt fiecare cîte o matrice pătratică de ordinul 2 a căror elemente sînt numere de tip *double*.

6. Fie declaraţia:

```
struct elev {
    char *nume;
    char *prenume;
    long nr_matricol;
    struct data_calendaristica
        data_nasterii;
    char *adresa;
    char *locul_nasterii;
    int clasa;
    int note[15];
} ;
```

Prin această declaraţie s-a introdus tipul *elev*. Declaraţia următoare defineşte date de tipul *elev:*

struct elev unelev,clasaelevi[25];

7. Ecranul pe care se afişează datele poate fi gestionat în mod *text* sau în mod *grafic*. În mod implicit, ecranul se gestionează în mod *text*. În acest mod, ecranul se compune, de obicei, din 25 linii a 80 de coloane.

Poziţia unui punct pe ecran se defineşte prin 2 valori (coordonate):

x - numărul coloanei

şi

y - numărul liniei.

Colţul din stînga sus are coordonatele:

x = 1;
y = 1.

Colţul din dreapta jos are coordonatele:

x = 80;
y = 25.

Pentru a controla afişarea datelor pe ecran, se pot utiliza diferite funcţii standard de gestiune a ecranului. O parte dintre aceste funcţii folosesc coordonatele *x şi y* de felul celor indicate mai sus.

Adesea este util să definim tipul *punct* printr-o declaraţie de felul următor:

```
struct punct {
    int x;
    int y;
};
```

Cu ajutorul acestui tip se pot defini poziţii pe ecran:

```
struct punct poz;
```

Structurile sînt date care se pot aloca la fel ca şi celelalte date(variabile simple sau tablouri). Astfel, o structură este *globală* dacă se declară în afara corpului oricărei funcţii, este *automatică* dacă se declară în corpul unei funcţii sau *statică* dacă declaraţia ei începe cu cuvîntul cheie *static*.

Datele elementare care sînt componente ale unei structuri se pot iniţializa. În acest scop, într-o declaraţie sau definiţie de structură, după numele structurii, se scrie caracterul= , iar după acesta construcţiile prin care se vor iniţializa componentele elementare ale structurii, separate prin virgulă şi incluse între acolade. Construcţiile utilizate la iniţializare sînt expresii constante care corespund elementelor pe care le iniţializează.

Exemple:

1.

```
struct data_calendaristica
 dc= { 1,"septembrie",1994};
```

unde:

data_calendaristica - Este tipul declarat în prealabil astfel:

```
struct data_calendaristica {
    int i;
    char luna[11];
    int an;
} ;
```

2. Fie tipul *complex*:

```
struct complex {
    double real;
    double imag;
};
```

Declaraţia de mai jos defineşte variabila complexă *z*, care iniţial are valoarea:

1 + 2i

struct complex z = { 1.0, 2.0 };

Componentele elementare neiniţializate ale unei structuri au valoarea iniţială *zero* dacă structura este *globală* sau *statică.*

În cazul structurilor *automatice*, elementele neiniţializate au o valoare iniţială *nedefinită.*

Operatorul unar adresă se poate aplica numelui unei structuri. Ca rezultat se obţine adresa primei sale componente elementare. De asemenea, se pot defini pointeri spre tipuri utilizator.

Exemplu:

```
struct complex {
    double real;
    double imag;
};
struct complex *p;
struct complex z;
...
p = &z;
```

10.2. Accesul la elementele unei structuri

Fie tipul utilizator *data_calendaristica* definit în paragraful precedent:

```
struct data_calendaristica {
    int zi;
    char luna[11];
```

```
        int an;
    };
```

şi variabila *dc* de tip *data_calendaristica*:

struct data_calendaristica dc;

Data *dc* are 3 componente:

zi, luna şi an - Sînt date elementare.

Se pune problema accesului (referirii) la componentele structurii *dc*.

Referirea la componentele unei stucturi se face utilizînd atît numele structurii cît şi a componentei respective. Aceasta se realizează printr-o construcţie de forma:

nume.nume1

unde:

nume - Este numele structurii.

nume1 - Este numele componentei la care se doreşte să se facă acces.

În exemplul de mai sus, vom avea acces la componenta *zi* a structurii *dc*, prin construcţia:

dc.zi

În mod analog:

dc.an

ne asigură accesul la componenta *an* a structurii *dc*. Datele *dc.zi şi dc.an* sînt de tip *int*.

Construcţia *dc.luna* este un pointer spre caractere şi permite accesul la componenta *luna* a structurii *dc*.

Punctul utilizat într-o construcţie de forma:

(1)nume.nume1

se consideră un *operator* şi el are prioritate *maximă*, la fel ca şi operatorii paranteză.

O construcţie de forma (1) se spune că este un *nume calificat*.

Exemple:

1.

```
struct data_calendaristica {
```

```
        int zi;
        char luna[11];
        int an;
    };
    struct data_calendaristica dc, d[10];
        ...
        dc.zi = 1;
        dc.an = 1995;
        strcpy(dc.luna,"septembrie");
        ...
        d[3].zi = dc.zi;
        d[3].an = dc.an;
        strcpy(d[3].luna,dc.luna);
        ...
```

Primele 3 instrucţiuni, atribuie structurii *dc* data calendaristică:

1 septembrie 1995

Ultimele 3, atribuie elementului d[3] valorile componentelor structurii *dc*, deci elementul respectiv păstrează aceeaşi dată calendaristică.

2.

```
    struct data_calendaristica {
        int zi;
        char luna[11];
        int an;
    };
    struct date_pers {
        char nume[50];
        char prenume[50];
        long cod;
        struct data_calendaristica
            data_nasterii,
            data_angajarii;
    };
    struct date_pers angajat,sectie[100];
    strcpy(angajat.nume,"Popescu");
    strcpy(angajat.prenume,"Gheorghe");
    angajat.cod = 123456789;
```

Pentru a defini data naşterii va trebui să folosim o dublă calificare:

```
angajat.data_nasterii.zi = 3;
strcpy(angajat.data_nasterii.luna,"august");
```

```
angajat.data_nasterii.an = 1970;
```

În mod analog se defineşte data angajării:

```
angajat.data_angajarii.zi = 15;
strcpy(angajat.data_angajarii.luna,"septembrie");
angajat.data_angajarii.an = 1993;
```

În limbajul C structurile nu se pot transfera prin parametri, ci numai pointerii spre ele. Aşa de exemplu, structura *angajat* poate fi prelucrată de funcţia *f*, dacă aceasta se apelează cu parametrul efectiv *&angajat* (adresa de început a zonei de memorie alocată structurii *angajat*):

```
f(&angajat);
```

Funcţia *f* are ca parametru un pointer spre tipul structurii *angajat*, adică spre tipul: *date_pers*.

Rezultă că funcţia *f* are antetul:

```
void f(struct date_pers *p)
```

Într-adevăr, declaraţia lui *p* specifică faptul că el are ca valoare adresa de început a unei zone de memorie în care se păstrează date de tip *date_pers*. Ori *&angajat* este chiar o astfel de adresă.

Problema care se pune în continuare este aceea a accesului la componentele structurii transferate prin adresa ei. În cazul de faţă se pune problema de a avea acces, în corpul funcţiei *f*, la componentele structurii *angajat*: nume, prenume etc.

În corpul funcţiei *f* nu se pot utiliza numele calificate:

angajat.nume, angajat.prenume, angajat.data_nasterii.zi etc.

deoarece nu este cunoscut numele structurii *angajat*. În locul numelui structurii se cunoaşte pointerul spre ea: *p*. De aceea, în construcţiile de mai sus nu avem decît să înlocuim numele *angajat* cu **p*. Se vor obţine construcţiile:

```
(*p).nume;
(*p).prenume;
(*p).data_angajarii.zi
etc.
```

Pentru a simplifica scrierea construcţiilor de acest fel s-a introdus o notaţie cu săgeată (-> minus urmat de caracterul mai mare) şi anume:

(*p). se înlocuieşte cu p ->

În felul acesta, construcţiile de mai sus se scriu mai simplu:

p -> nume;
p -> prenume;
p -> data_angajarii.zi etc.

Săgeata, ca şi punctul, se consideră un operator de prioritate *maximă*, la fel ca şi parantezele. Amintim că aceşti operatori se asociază de la stînga spre dreapta.

În concluzie, accesul la componentele unei structuri se realizează fie printr-o construcţie de forma:

nume.nume1

fie prin:

pointer -> *nume1*

unde:

nume - Este numele structurii.
nume1 - Este numele componentei.
pointer - Este un pointer spre structură.

10.3. Asignări de nume pentru tipuri de date (declaraţii de tip)

Tipurile predefinite se identifică printr-un cuvînt cheie. În cazul tipurilor utilizator în locul cuvîntului cheie se utilizează construcţia:

struct nume

unde:

nume - Este introdus printr-o declaraţie de forma:

struct *nume* {
...
};

În limbajul C se poate atribui un nume unui tip, indiferent că el este un tip predefinit sau unul utilizator.

În acest scop se utilizează o construcţie de forma:

(1) **typedef** *tip nume_tip*;

unde:

tip - Este fie un tip predefinit, fie un tip utilizator.
nume_tip - Este numele care se atribuie tipului definit de *tip*.

După ce s-a atribuit un nume unui tip, numele respectiv poate fi utilizat pentru a declara date de acel tip, exact la fel cum se utilizează în declaraţii cuvintele cheie ale tipurilor predefinite: *int, char, float* etc.

De obicei, numele atribuit unui tip se scrie cu litere mari. Construcţia (1) de mai sus o vom considera ca fiind o declaraţie prin care se atribuie un nume unui tip sau mai scurt *declaraţie de tip*.

Exemple:

1. Fie declaraţia:

```
typedef int INTREG;
```

După această declaraţie, cuvîntul INTREG se poate utiliza pentru a defini date de tip *int*. Deci declaraţia:

```
INTREG x;
```

este identică cu declaraţia:

```
int x;
```

2. Fie declaraţia:

```
typedef float REAL;
```

Datele:

```
REAL a,b;
```

sînt de tip *float*.

3.

```
typedef struct data_calendaristica {
    int zi;
    char luna[11];
    int an;
} DC;
```

Declaraţia:

```
DC data_nasterii, data_angajarii;
```

este identică cu declaraţia:

```
struct data_calendaristica data_nasterii, data_angajarii;
```

4.

```
typedef struct {
    double real;
```

```
    double imag;
} COMPLEX;
```

În continuare se pot declara numere complexe prin declaraţii de forma:

COMPLEX z, tz[10];

5.

```
typedef struct {
    int x;
    int y;
} PUNCT;
PUNCT a,b;
```

Variabilele *a şi b* sînt structuri de tip PUNCT. Fiecare are o componentă x (abscisa) şi una y (ordonata).

6. Un dreptunghi se poate defini prin două vîrfuri diametral opuse ale sale. De obicei, se consideră vîrful din colţul stînga sus şi cel din dreapta jos.

Definim tipul utilizator DREPTUNGHI:

```
typedef struct {
    PUNCT stanga_sus;
    PUNCT dreapta_jos;
} DREPTUNGHI;
```

În continuare se pot defini date de tip DREPTUNGHI:

DREPTUNGHI d, td[10];

7. typedef int *PINT;

PINT este numele atribuit pentru tipul pointer spre *int*.

Declaraţia:

PINT p, t[100];

este identică cu declaraţia:

int *p, *t[100];

Construcţia:

tip (**p*) (*lista de tipuri*)

se poate folosi în antetul unei funcţii şi cu ajutorul ei, parametrul *p* se declară ca fiind pointer spre o funcţie care returnează o valoare de tipul *tip*, iar *lista de tipuri* defineşte tipurile parametrilor funcţiei spre care pointează *p*. La tipul parametrului *p* de mai sus se poate atribui un nume printr-o construcţie *typedef* de forma:

(2) **typedef** *tip* (**nume_tip*)(*lista tipurilor*);

Exemple:

1. Fie funcţia *f* de antet:

   void f(int (*p)(float,char,long,int))

   Declaraţia:

   typedef int (*PF) (float,char,long,int);

ne permite să rescriem mai simplu antetul funcţiei *f*:

   void f(PF p)

2. Folosind declaraţia:

   typedef double (* PDF) (double);

   putem rescrie antetul funcţiei *itrapez* din exerciţiul 8.25, în felul următor:

   double itrapez(double a, double b, int n, PDF p)

Din cele de mai sus se observă că declaraţiile *typedef* ne permit adesea să facem simplificări la scrierea antetelor şi prototipurilor funcţiilor. Acestea devin substanţiale mai ales atunci cînd apar expresii complicate cu pointeri.

Sensul unei expresii cu pointeri se poate stabili dacă se exprimă prin cuvinte efectul fiecărui operator, ţinînd seama de regulile de prioritate şi asociativitate.

În lucrarea [5] se dau exemple de expresii complexe cu pointeri şi se stabilesc sensurile lor înlocuind prin cuvinte efectele operatorilor pe care-i conţin.

Un astfel de exemplu indicat în lucrarea [5] este următoarea declaraţie:

int *(*(*x)[6]) ( ) ;

Sensul lui *x* se determină astfel:

int * defineşte tipul pointer spre *int*, deci:

a. (*(*x)[6]) () - Este pointer spre *int*.

b. (*(*x)[6]) - Este o funcţie şi ţinînd seama de afirmaţia de la punctul a., rezultă că această funcţie returnează un pointer spre *int*.

c. *(*x)[6] - Are acelaşi sens cu cel de la punctul b.

d. (*x)[6] - Pointer spre construcţia definită precedent, deci pointer spre o funcţie care returnează un pointer

spre întregi.

e. (*x) - Tablou de 6 elemente, fiecare element este un pointer spre o funcţie care returnează un pointer spre întregi.

f. *x - Construcţie identică cu cea de la punctul e.

g. x - Este pointer spre un tablou de 6 elemente, fiecare element fiind un pointer spre o funcţie care returnează un pointer spre *int*.

Un alt exemplu din aceeaşi lucrare este declaraţia:

char (*(*y())[]) ( );

Această declaraţie se interpretează astfel:

a. (*(*y())[])() - Defineşte o valoare de tip *char*.

b. (*(*y())[]) - Este o funcţie care returnează o valoare de tip *char*.

c. *(*y())[] - Este aceeaşi construcţie ca şi cea de la punctul b.

d. (*y())[] - Este un pointer spre o funcţie care returnează o valoare de tip *char*.

e. (*y()) - Este un tablou de pointeri spre funcţii care returnează o valoare de tip *char*.

f. *y() - Este aceeaşi construcţie ca şi cea de la punctul e.

g. y() - Este pointer spre un tablou de pointeri spre funcţii care returnează o valoare de tip *char*.

h. y - Este o funcţie care returnează un pointer spre un tablou de pointeri spre funcţii care returnează o valoare de tip *char*.

Declaraţiile de acest gen pot fi simplificate folosind declaraţii de tip.

De exemplu, în lucrarea [5] se declară tipuri intermediare pentru a simplifica declaraţia:

int *(*(*x)[6])();

analizată mai sus.

Exerciţii:

10.1 Să se scrie o funcţie care calculează şi returnează modulul unui număr complex.

Dacă:

z = x + iy

atunci modulul numărului complex este rădăcina pătrată din:

x*x + y*y

FUNCŢIA BX1

```
double dmodul (COMPLEX *z)
/* calculeaza si returneaza modulul numarului complex z */
{
 return sqrt(z -> x * z -> x + z -> y * z -> y);
}
```

10.2 Să se scrie o funcţie care calculează şi returnează argumentul unui număr complex.

Dacă:

z = x + iy

atunci

arg z

se calculează astfel:

a. Dacă x = y = 0, arg z = 0.

b. Dacă y = 0 şi x este diferit de zero,

atunci

dacă x > 0, arg z =0;

altfel

arg z = pi = 3.14159265358979.

c. Dacă x = 0 şi y este diferit de zero,

atunci

dacă y > 0, arg z = pi/2;

altfel

arg z = 3*pi/2.

d. Dacă x şi y sînt diferiţi de zero,

atunci fie:

a = arctg(y/x)

d1. Dacă x > 0 şi y > 0, atunci arg z = a.

d2. Dacă x > 0 şi y < 0, atunci arg z = 2*pi + a.

d3. Dacă x < 0 şi y > 0, atunci arg z = pi + a.

d4. Dacă x < 0 şi y < 0, atunci arg z = pi + a.

Se observă că pentru x < 0 şi y diferit de zero:

arg z = pi + a;

altfel, dacă x > 0 şi y < 0, atunci

arg z = 2*pi + a

FUNCŢIA BX2

```
double darg(COMPLEX *z)
{
 double a;

 if( z-> x == 0 && z->y == 0)
     return 0.0;
 if( z->y == 0)
     if( z->x > 0)
         return 0.0;
     else /* y=0 si x <0 */
         return PI;
 if(z->x == 0)
     if(z->y > 0)
         return PI/2;
     else /* x =0 si y<0 */
         return (3*PI)/2;

/* x != 0 si y != 0 */
 a = atan(z->y/z->x);
 if(z->x <0 ) /* x < 0 si y != 0 */
     return a + PI;
 else /* x > 0 */
     if( z->y < 0 ) /* x > 0 si y < 0 */
         return 2*PI + a;

/* x > 0 si y > 0 */
 return a;
}
```

10.3 Să se scrie un program care citeşte numere complexe şi le afişează împreună cu modulul şi argumentul lor.

PROGRAMUL BX3

```
#include <stdio.h>
#include <math.h>

#define PI 3.14159265358979

typedef struct {
 double x;
 double y;
} COMPLEX;

#include "bx1.cpp"
#include "bx2.cpp"

main() /* citeste numere complexe si le afiseaza impreuna cu modulul
          si argumentul corespunzator */
{
 COMPLEX complex;

 while(scanf("%lf %lf",
          &complex.x,&complex.y) == 2 ){
     printf("a+ib= %g + i*(%g)\n",complex.x,
            complex.y);
     printf("modul=%g\targ=%g\n",
           dmodul(&complex),darg(&complex));
 }
}
```

10.4 Să se scrie o funcţie care afişează un caracter într-un punct de coordonate date.

Ecranul, în mod text, poate fi gestionat folosind funcţii care au prototip în fişierul *conio.h*. Astfel, pentru a poziţiona cursorul în punctul de coordonate (x,y), se va apela funcţia *gotoxy*. Prototipul acestei funcţii este:

void gotoxy (int *x*, **int** *y*);

Pentru a afişa, începînd cu punctul poziţionat cu ajutorul funcţiei *gotoxy*, se pot folosi diferite funcţii, ca de exemplu, *putch* sau *cprintf* în locul funcţiei *printf*. Funcţia *cprintf* are aceeaşi parametri ca şi funcţia *printf*. Prototipul ei se află în fişierul *conio.h* spre deosebire de al funcţiei *printf* care se află în

stdio.h.

O diferenţă a funcţiei *cprintf* faţă de funcţia *printf* constă în aceea că '\n' se interpretează diferit. Astfel, apelul:

```
cprintf("\n");
```

trece cursorul pe rîndul următor lăsîndu-l în aceeaşi coloană. De aceea, apelul:

```
printf("\n");
```

se echivalează cu apelul:

```
cprintf("\n\r");
```

dacă cursorul nu este pe ultimul rînd (25).

Funcţia *cprintf* nu realizează defilarea ecranului şi de aceea, caracterul '\n' se neglijează dacă cursorul se află pe ultimul rînd.

FUNCŢIA BX4

```
void afiscar(PUNCT *p, char c)
/* - afiseaza caracterul c in punctul de coordonate (p->x,p->y);
   - se presupune ca punctul nu este in afara ecranului. */
{
 gotoxy( p->x, p ->y );
 putch(c);
}
```

10.5 Să se scrie o funcţie care verifică dacă un punct are coordonatele în limitele ecranului(25 de linii a 80 coloane).

Funcţia returnează valoarea 1 dacă punctul se află în limitele ecranului şi 0 în caz contrar.

FUNCŢIA BX5

```
int limecr(PUNCT *p)
/* returneaza 1 daca punctul de coordonate (p->x,p->y) se afla in
   limitele ecranului si zero altfel */
{
 if(p->x <= 0 || p->x > 80)
    return 0;
 if(p->y <= 0 || p->y > 25)
    return 0;
 return 1;
```

```
}
```

10.6 Să se scrie o funcţie care trasează un segment de dreaptă orizontal afişînd repetat un acelaşi caracter dat.

Se presupune că limitele segmentului se află pe ecran.

FUNCŢIA BX6

```
void segoriz(PUNCT *a,PUNCT *b,char c)
/* - traseaza un segment de dreapta orizontal prin afisarea repetata a
     caracterului c;
   - se presupune ca punctul a precede pe b si ca a->y = b->y */
{
 int i;
 PUNCT crt;

 crt.y = a ->y;
 for(i=a->x; i <= b->x; i++){
     crt.x = i;
     afiscar(&crt,c);
 }
}
```

10.7 Să se scrie o funcţie care trasează un segment de dreaptă vertical afişînd repetat un acelaşi caracter dat.

Se presupune că limitele segmentului se află pe ecran.

FUNCŢIA BX7

```
void segvert(PUNCT *a,PUNCT *b,char c)
/* - traseaza un segment de dreapta vertical prin afisarea repetata a
     caracterului c;
   - se presupune ca punctul a este deasupra punctului b si ca
     a->x = b->x */
{
 int i;
 PUNCT crt;

 crt.x = a->x;
 for(i = a->y; i <= b->y; i++) {
     crt.y = i;
     afiscar(&crt,c);
```

```
   }
 }
```

10.8 Să se scrie o funcţie care trasează un segment de dreaptă oblic de pantă 45 de grade sexagesimale afişînd repetat un acelaşi caracter dat.

FUNCŢIA BX8

```
void segoblic(PUNCT *a,PUNCT *b,char c)
/* - traseaza un segment de dreapta oblic de panta 45 grade sexagesimale
     prin afisarea repetata a caracterului c;
   - se presupune ca punctele a si b permit trasarea unui astfel de seg-
     ment de dreapta si ca ele au coordonate ce apartin ecranului. */
{
 int i,j,k,l;
 int pasi,pask;

 i = a->x;
 j = b->x;
 k = a->y;
 l = b->y;
 if( i < j) /* a in stinga lui b */
    pasi = 1;
 else /* a in dreapta lui b */
    pasi = -1;
 if( k < l) /* a deasupra lui b */
    pask = 1;
 else /* b deasupra lui a */
    pask = -1;

/* trasarea segmentului */
 for( ; (j-i)*pasi >= 0; i +=pasi, k += pask) {
    gotoxy(i,k);
    putch(c);
 }
}
```

Observaţie:

Segmentul se trasează de la punctul *a* spre punctul *b*. De exemplu, dacă punctul *a* este deasupra lui *b* (pask = 1)şi în dreapta lui *b* (pasi = -1), atunci se observă că *i* descreşte pînă cînd *i* < *j*. În acest moment produsul *(j-i)*pasi<0* şi deci ciclul *for* se termină.

De fiecare dată cînd *i* s-a micşorat cu o unitate, *k* a crescut cu o unitate. De aceea, poziţia caracterului curent faţă de cel precedent se află în coloana din stînga şi cu o linie mai jos.

Punctele *a* şi *b* trebuie să satisfacă relaţia:

(j-i)*pasi = (l-k)*pask

şi de aceea, cînd $i=j$, vom avea şi $k=l$, adică se ajunge în punctul *b*.

10.9 Să se scrie un program care trasează un dreptunghi.

Laturile orizontale se trasează folosind caracterul "-" (minus), iar cele verticale folosind caracterul "|" (bara verticală).

PROGRAMUL BX9

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

typedef struct {
 int x;
 int y;
} PUNCT;

#include "bx4.cpp"      /* afiscar      */
#include "bx6.cpp"      /* segoriz      */
#include "bx7.cpp"      /* segvert      */
#include "bviii2.cpp"  /* pcit_int     */
#include "bviii3.cpp"  /* pcit_int_lim */

main() /* citeste coordonatele coltului stinga sus si a coltului dreapta
          jos ale unui dreptunghi dupa care traseaza dreptunghiul
          respectiv folosind caracterul
                   -    pentru laturile orizontale si
                   |    pentru cele verticale. */
{
 PUNCT v1,v2,v;
 char er[] = "s-a tastat EOF\n";

/* citeste coordonatele celor doua colturi ale dreptunghiului */
/* coltul din stinga sus */
 if(pcit_int_lim("x_stanga_sus: ",1,79,
        &v1.x) == 0 ) {
```

```
      printf(er);
      exit(1);
  }
  if(pcit_int_lim("y_stanga_sus: ",1,24,
          &v1.y) == 0 ) {
      printf(er);
      exit(1);
  }

/* coltul din dreapta jos */
  if(pcit_int_lim("x_dreapta_jos: ",v1.x+1,80,
          &v2.x) == 0 ) {
      printf(er);
      exit(1);
  }
  if(pcit_int_lim("y_dreapta_jos: ",v1.y+1,25,
          &v2.y) == 0 ) {
      printf(er);
      exit(1);
  }

/* sterge ecranul */
  clrscr();

/* trasarea laturilor dreptunghiului */
/* laturile orizontale */
/* latura de sus */
  v.x = v2.x;
  v.y = v1.y;
  segoriz(&v1,&v,'-');

/* latura de jos */
  v.x = v1.x;
  v.y = v2.y;
  segoriz(&v, &v2,'-');

/* laturile verticale */
  v1.y++;
  v2.y--;
  if(v1.y > v2.y )
      exit(0); /* nu sint laturi verticale */

/* latura din stinga */
```

```
 v.x = v1.x;
 v.y = v2.y;
 segvert(&v1,&v,'|');

/* latura din dreapta */
 v.x = v2.x;
 v.y = v1.y;
 segvert(&v,&v2,'|');
 getch();
}
```

10.10 Să se scrie un program care afişează figura de mai jos.

```
     *
    ***
   * * *
  *  *  *
 *   *   *
***********
**   *   **
* *  *  * *
*  * * *  *
*   ***   *
***********
*   ***   *
*  * * *  *
* *  *  * *
**   *   **
***********
 *   *   *
  *  *  *
   * * *
    ***
     *
```

Această figură se compune din segmente de dreaptă orizontale, verticale şi oblice de pantă 45 de grade sexagesimale. De aceea, ea se poate afişa folosind funcţiile *segoriz, segvert şi segoblic.*

Vîrful de sus are coordonatele A(40,1). Segmentul vertical din stînga are extremităţile B(35,6) şi C(35,16).

Vîrful de jos are coordonatele D(40,21). Segmentul vertical din dreapta are extremităţile F(45,6) şi E(45,16).

Segmentul orizontal de sus are extremităţile B şi F, iar cel de jos extremităţile C şi E.

Segmentul orizontal, care este axă de simetrie a figurii, are extremităţile G(35,11) şi H(45,11).

În afară de aceste segmente, figura mai conţine următoarele segmente oblice:

AB, AF, BE, FC, CD şi ED.

PROGRAMUL BX10

```
#include <conio.h>

typedef struct {
 int x;
 int y;
} PUNCT;

#include "bx4.cpp" /* afiscar */
#include "bx6.cpp" /* segoriz */
#include "bx7.cpp" /* segvert */
#include "bx8.cpp" /* segoblic */

#define C '*'

main () /* afiseaza o figura formata din segmente orizontale, verticale
           si oblice folosind caracterul *. */
{
 static PUNCT a = {40,1};
 static PUNCT b = {35,6};
 static PUNCT c = {35,16};
 static PUNCT d = {40,21};
 static PUNCT e = {45,16};
 static PUNCT f = {45,6};
 static PUNCT g = {35,11};
 static PUNCT h = {45,11};

/* sterge ecranul */
 clrscr();

/* afiseaza segmentele verticale */
 segvert(&b,&c,C);
 segvert(&a,&d,C);
```

```
  segvert(&f,&e,C);

 /* afiseaza segmentele orizontale */
  segoriz(&b,&f,C);
  segoriz(&c,&e,C);
  segoriz(&g,&h,C);

 /* afiseaza segmentele oblice */
  segoblic(&a,&b,C);
  segoblic(&a,&f,C);
  segoblic(&b,&e,C);
  segoblic(&f,&c,C);
  segoblic(&c,&d,C);
  segoblic(&e,&d,C);
  getch();
}
```

10.11 Să se scrie un program care realizează următoarele:

- citeşte datele de identificare şi notele obţinute de o grupă de candidaţi la admiterea în învăţămîntul superior;
- listează datele citite împreună cu media notelor obţinute de fiecare candidat, media grupei la fiecare materie şi media mediilor.

La începutul programului se citeşte un întreg ce reprezintă numărul examenelor susţinute, apoi se citesc denumirile materiilor la care s-au susţinut examene. În continuare, pentru fiecare candidat, se citesc:

- nume;
- prenume;
- data naşterii;
- adresa;
- nota obţinută la fiecare examen.

Dacă un candidat are mai multe prenume, acestea se concatenează prin liniuţa de unire, nu se vor separa prin spaţii. În felul acesta, chiar dacă sînt mai multe prenume, ele se pot citi prin *scanf*, ca şi cum ar fi un singur prenume.

Data naşterii se tastează printr-un şir de 6 cifre:

zzllaa

deci, primele două cifre reprezintă ziua, următoarele două luna, iar ultimele

două anul.

Adresa este o succesiune de caractere care nu conţine caractere albe. Spaţiile din cadrul unei adrese se vor înlocui prin caracterul subliniere. În felul acesta, adresa poate fi citită ca un singur element.

Programul utilizează următoarele funcţii definite în exerciţiile precedente:

pcit_int_lim	- Pentru a citi date de tip *int* care se află într-un interval precizat (exerciţiul 8.3).
pcit_data_calend	- Pentru a citi şi valida o dată calendaristică (exerciţiul 8.4).
pcit_int	- Funcţie apelată de *pcit_int_lim* (exerciţiul 8.2).
v_calend	- Funcţie apelată de *pcit_data_calend* (exerciţiul 6.5).

Programul mai apelează funcţia *pcit_sir* definită în acest exerciţiu, care realizează următoarele:

- afişează un text explicativ;
- citeşte o succesiune de caractere;
- păstrează succesiunea citită în memoria *heap*;
- returnează pointerul spre începutul zonei din memoria *heap* în care se păstrează caracterele respective.

Programul se execută conform paşilor de mai jos:

1. Citeşte pe *n*, numărul examenelor(se apelează *pcit_int_lim*).
2. Citeşte denumirile celor *n* materii care se examinează (se apelează *pcit_sir* de n ori).
3. i = 0 (numarător pentru candidaţi).
4. Se citesc datele de identificare ale unui candidat şi notele acestuia.

   Se păstrează datele citite, apoi se trece la pasul 5.

   Citirile se realizează apelînd funcţiile indicate mai sus.

   Dacă nu mai sînt date de citit, se trece la punctul 7.
5. i = i+1.
6. Se reia de la punctul 4.
7. Se fac iniţializări pentru calcule de medii pe materii şi a mediei generale.
8. Se afişează antetul listei.

9. Pentru fiecare candidat se afişează:

   - lista cu datele de identificare a candidatului, notele obţinute la fiecare materie (nota 0 se consideră neprezentare la examen) şi media notelor respective;
   - după ce s-au listat toate datele pentru toţi candidaţii, se trece la punctul 10.

10. Se afişează mediile pe materii;

11. Se afişează media generală.

În programul de faţă se presupune că sînt cel mult 5 examene.

Datele relative la un candidat se păstrează în memoria *heap*. De asemenea, tot în memoria *heap* se păstrează denumirile materiilor de examinat.

PROGRAMUL BX11

```
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
#include <string.h>

#include "bviii2.cpp" /* pcit_int        */
#include "bviii3.cpp" /* pcit_int_lim    */
#include "bvi5.cpp"   /* v_calend        */
#include "bviii4.cpp" /* pcit_data_calend */

#define MAX 100

int nrzile[] = {0,31,28,31,30,31,30,31,
                31,30,31,30,31};
int pcit_sir(char *, char **);

main() /* listeaza situatia notelor unei grupe de candidati obtinute la
          admiterea in invatamintul superior */
{
 typedef struct {
     int zi;
     int luna;
     int an;
 } DC;
```

```
typedef struct {
    char *nume;
    char *prenume;
    DC data_nasterii;
    char *adresa;
    int nrex;
    int nota[5];
} CAND;

CAND *tcand[MAX];
char *denex[5];
int nex;
char er[]="s-a citit EOF\n";
int i,j,k;
char *temp;
float tmed[5];
float medgen;
float med;
```

/* citeste numarul examenelor */

```
if(pcit_int_lim("numarul examenelor:1-5: ",1,5,
                &nex) == 0 ){
    printf(er);
    exit(1);
}
```

/* citeste denumirile materiilor si le pastreaza in memoria heap */

```
for(i=0; i<nex; i++)
    if(pcit_sir("denumire materie: ",
          &denex[i]) == 0 ) {
        printf(er);
        exit(1);
    }
```

/* se citesc datele de identificare ale elevilor si notele */

```
i = 0;
while( i < MAX ) { /* 1 */
    printf("se citesc datele elevului nr: %d\n",
           i+1);
```

/* citeste numele candidatului */

```
    if(pcit_sir("nume: ",&temp) == 0)
        break; /* nu mai sint alti candidati */
```

/* rezerva zona pentru o data de tip CAND */
```
    if((tcand[i]=(CAND *)malloc(sizeof(CAND)))
              == 0 ) {
        printf("memorie insuficienta\n");
        exit(1);
    }
```

/* tcand[i] are ca valoare adresa de inceput a zonei de memorie din memoria heap in care se pastreaza o data structurata de tip CAND */

/* se pastreaza pointerul spre numele candidatului curent citit, nume pastrat de functia pcit_sir in memoria heap */
```
    tcand[i] -> nume = temp;
```

/* se citeste prenumele */
```
    if(pcit_sir("prenume: ",
              &tcand[i] -> prenume)== 0 ) {
        printf(er);
        exit(1);
    }
```

/* citeste data nasterii */
```
    if(pcit_data_calend(
             &tcand[i] -> data_nasterii.zi,
             &tcand[i] -> data_nasterii.luna,
             &tcand[i] -> data_nasterii.an) == 0) {
        printf(er);
        exit(1);
    }
```

/* citeste adresa */
```
    if(pcit_sir("adresa: ",
              &tcand[i] -> adresa)== 0 ) {
        printf(er);
        exit(1);
    }
    tcand[i] -> nrex = nex;  /* pastreaza numarul exame-
                                nelor */
```

/* se citesc notele; se tasteaza in ordinea citirii denumirilor materiilor de examinat */
```
    for(j=0; j < nex; j++)
```

```
        if(pcit_int_lim(denex[j],0,10,
             &tcand[i] -> nota[j])==0 ){
            printf(er);
            exit(1);
        }
```

/* s-au citit toate datele pentru un candidat */

```
    i++;
  } /* sfirsit while 1 */
```

/* initializari pentru calculul mediilor pe materii */

```
  for(j=0; j < nex; j++ )
      tmed[j] = 0.0;
  medgen = 0.0;
```

/* afiseaza antetul */

```
  printf("\n\n\tNotele de la examenul de admitere\
         din 7 septembrie 1994\n");
  printf("\n\n\n");
  printf("%20s"," "); /* 20 spatii pentru nume prenume */
```

/* se scriu denumirile examenelor */

```
  for(j=0; j<nex;j++) {
      printf("%10s",denex[j]);
      printf(" " );/* cel putin un spatiu dupa fiecare denumire */
  }
  printf("Media\n\n");
```

/* lista cu datele de identificare, notele si media pentru
fiecare candidat */

```
  for(j=0; j<i; j++) {
```

/* datele de identificare */

```
      printf("%25s %25s %02d/%02d/%d\n",
           tcand[j] -> nume,
           tcand[j] -> prenume,
           tcand[j] -> data_nasterii.zi,
           tcand[j] -> data_nasterii.luna,
           tcand[j] -> data_nasterii.an);
```

/* adresa */

```
      printf("\t%s\n",tcand[j] -> adresa);
```

```
/* note */
    printf("%20s"," ");
    for(k=0, med = 0.0; k < nex; k++) {
        printf("%10d ",tcand[j] -> nota[k]);
                   med += tcand[j] -> nota[k];
                    tmed[k] += tcand[j] -> nota[k];
    }
    printf("%6.2f\n",med/nex);
 } /* sfirsit date candidati */

/* afisare media pe materii */
 printf("\n\n Media pe materii ");
 for(j=0;j<nex;j++) {
    printf("%11.2f", tmed[j]/i);
    medgen += tmed[j] / i;
 }

/* afisare media generala */
 printf("\n\n Media generala: %10.2f\n",
        medgen / nex);
} /* sfirsit main */

int pcit_sir( char *text,char **sir)
/* - afiseaza text si citeste o succesiune de caractere;
  - pastreaza succesiunea citita in memoria heap si atribuie adre-
   sa acestei zone, pointerului sir;
  - returneaza:
        0 - la sfirsit de fisier
        1 - in caz contrar
  - intrerupe executia programului daca nu exista memorie heap
   suficienta pentru a pastra sirul de caractere citit. */
{
 char t[255];
 char *p;

 printf(text);
 if(gets(t) == 0 )
    return 0;

/* rezerva zona in memoria heap */
 if((p = (char *)malloc(strlen(t) + 1 ))== 0 ) {
    printf("memorie insuficienta\n");
    exit(1);
```

```
}
/* transfera sirul din t la adresa continuta in p */
strcpy(p,t);
*sir = p;
return 1;
}
```

10.12 Să se scrie o funcţie care schimbă semnul la partea reală şi cea imaginară a unui număr complex.

Funcţia utilizează tipul COMPLEX introdus prin declaraţia:

```
typedef struct {
    double x;
    double y;
} COMPLEX;
```

Acest tip se utilizează în exerciţiile care urmează în acest capitol.

FUNCŢIA BX12

```
void negcomplex(COMPLEX *a,COMPLEX *b) /* b = -a */
{
 b -> x = - a -> x;
 b -> y = - a -> y;
}
```

10.13 Să se scrie o funcţie care atribuie partea reală şi imaginară a unui număr complex la partea reală şi imaginară a unui alt număr complex.

FUNCŢIA BX13

```
void atribcomplex(COMPLEX *a,COMPLEX *b) /* b = a */
{
 b ->x = a -> x;
 b -> y = a -> y;
}
```

10.14 Să se scrie o funcţie care adună două numere complexe.

FUNCŢIA BX14

```
void adcomplex(COMPLEX *a, COMPLEX *b,COMPLEX *c)
/* c = a+b */
```

```
{
 c -> x = a -> x + b -> x;
 c -> y = a -> y + b -> y;
}
```

10.15 Să se scrie o funcţie care scade două numere complexe.

FUNCŢIA BX15

```
void sccomplex(COMPLEX *a,COMPLEX *b,COMPLEX *c)
/* c = a - b */
{
 c -> x = a -> x - b -> x;
 c -> y = a -> y - b -> y;
}
```

10.16 Să se scrie o funcţie care înmulţeşte două numere complexe.

Dacă *a şi b* sînt două numere complexe de forma:

a = ax + i*ay

şi

b = bx + i*by

atunci produsul lor se defineşte astfel:

a*b = (ax + i*ay)*(bx + i*by) = ax*bx - ay*by + (ax *by + bx*ay)*i

FUNCŢIA BX16

```
void mulcomplex(COMPLEX *a,COMPLEX *b,COMPLEX *c)
/* c = a*b */
{
 c -> x = a -> x * b -> x - a ->y * b -> y;
 c -> y = a -> x * b -> y + b -> x * a -> y;
}
```

10.17 Să se scrie o funcţie care împarte două numere complexe. Ea returnează valoarea zero dacă împărţitorul este nul şi unu în caz contrar.

Dacă *a şi b* sînt două numere complexe de forma:

a = ax + i*ay

şi

b = bx + i*by

atunci:

c = a/b = (ax +i*ay)/(bx + i*by)=
= [(ax + i*ay)(bx - i*by)]/(bx*bx + by*by) =
=(ax *bx + ay *by)/(bx *bx + by *by) +
i * (ay * bx - ax * by)/(bx * bx + by * by)

FUNCŢIA BX17

```
int divcomplex(COMPLEX *a,COMPLEX *b,COMPLEX *c)
/* c = a/b */
{
 double d;

 d = b->x * b->x + b->y * b->y;
 if( d == 0 )
    return 0;
 c->x = (a->x* b->x + a->y * b->y)/d;
 c->y = (a->y* b->x - a->x* b->y)/d;
 return 1;
}
```

10.18 Să se scrie o funcţie care afişează un text şi citeşte un număr de tip *DOUBLE*.

Funcţia returnează zero dacă se întîlneşte sfîrşitul de fişier şi unu în caz contrar.

Această funcţie este similară cu funcţia *pcit_int* definită în exerciţiul 8.2.

FUNCŢIA BX18

```
int pcit_double(char text[], double *x)
/* - afiseaza caracterele pastrate in tabloul text, citeste un numar de tip
     double si-l pastreaza in zona de memorie spre care pointeaza x;
   - returneaza:
        0 - la intilnirea sfirsitului de fisier;
        1 - altfel. */
{
 char t[255];

 for ( ; ; ) {
    printf(text);
```

```
      if(gets(t) == 0 )
         return 0;
      if(sscanf(t,"%lf",x) == 1)
         return 1;
      printf("nu s-a tastat un numar\n");
   }
}
```

10.19 Să se scrie o funcţie care citeşte partea reală şi partea imaginară a unui număr complex.

Funcţia returnează zero la întîlnirea sfîrşitului de fişier şi unu în caz contrar.

FUNCŢIA BX19

```
int pcitcomplex(COMPLEX *a)
/* - citeste partea reala si cea imaginara a numarului complex a;
   - returneaza:
         0 - la intilnirea sfirsitului de fisier;
         1 - altfel. */
{
 if(pcit_double("partea reala: ", &a -> x)== 0 )
    return 0;
 if(pcit_double("partea imaginara: ",&a -> y)== 0 )
    return 0;
 return 1;
}
```

10.20 Să se scrie un program care citeşte trei numere complexe *a,b,c* care sînt coeficienţii ecuaţiei:

a*x*x + b*x + c = 0

rezolvă şi afişează rădăcinile ecuaţiei respective.

PROGRAMUL BX20

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define PI 3.14159265358979
```

```
typedef struct {
 double x;
 double y;
} COMPLEX;

#include "bx12.cpp" /* negcomplex */
#include "bx14.cpp" /* adcomplex  */
#include "bx15.cpp" /* sccomplex  */
#include "bx16.cpp" /* mulcomplex */
#include "bx17.cpp" /* divcomplex */
#include "bx1.cpp"  /* dmodul     */
#include "bx2.cpp"  /* darg       */
#include "bx18.cpp" /* pcit_double */
#include "bx19.cpp" /* pcitcomplex */

main() /* - citeste pe a,b,c - numere complexe;
          - rezolva si afiseaza radacinile ecuatiei:
                a*x*x + b*x + c = 0. */
{
 COMPLEX a,b,c,x1,x2,bp,temp,aa,temp1;
 char er[] = "s-a tastat EOF\n";
 double r,arg;
 COMPLEX patru = {4.0,0.0};

/* se citesc coeficientii*/

 if(pcitcomplex(&a) == 0 ) {
     printf(er);
     exit(1);
 }
 if(pcitcomplex(&b) == 0 ) {
     printf(er);
     exit(1);
 }
 if(pcitcomplex(&c) == 0 ) {
     printf(er);
     exit(1);
 }
 if( a.x == 0 && a.y == 0 && b.x == 0 &&
     b.y == 0 && c.x == 0 && c.y == 0 ) {

/* a = b = c = 0 */
     printf("ecuatie nedeterminata\n");
```

```
        exit(0);
    }
    if( a.x == 0 && a.y == 0 && b.x == 0 &&
        b.y == 0 ) {

/* a = b = 0 si c != 0 */
        printf("ecuatia nu are solutie\n");
        exit(1);
    }
    if(a.x == 0 && a.y == 0 ) {

/* ecuatie de gradul 1 */
        negcomplex(&c,&x1);
        divcomplex(&x1,&b,&x2);
        printf("ecuatie de gradul 1\n");
        printf("x=%g + i*(%g)\n", x2.x, x2.y);
        exit(0);
    } /* temp = b*b - 4*a*c */

    mulcomplex(&b,&b,&bp);
    mulcomplex(&patru,&a,&temp);
    mulcomplex(&temp,&c,&temp1);
    sccomplex(&bp,&temp1,&temp);

/*temp = x + i*y
  r = modul(temp)
  arg = argument(temp) */

    r = dmodul(&temp);
    arg = darg(&temp);

/* sqrt(temp); temp = r*(cos(arg) + i*sin(arg));
   sqrt(temp)=sqrt(r)*(cos(arg/2)+i*sin(arg/2)) */

    r = sqrt(r);
    arg = arg/2;

/* sqrt(b*b - 4*a*c) = r*(cos(arg)+i*sin(arg)) */

    temp.x = r*cos(arg);
    temp.y = r*sin(arg);
```

```
/* aa = 2*a = a+a */

 adcomplex(&a,&a,&aa);

/* x1=(-b+sqrt(b*b-4*a*c))/(2*a) */
/* bp=-b */

 negcomplex(&b,&bp);
 adcomplex(&bp,&temp,&templ);
 divcomplex(&templ,&aa,&x1);

/* x2=(-b-sqrt(b*b-4*a*c))/(2*a) */

 sccomplex(&bp,&temp,&templ);
 divcomplex(&templ,&aa,&x2);
 printf("x1=%g+i(%g)\n", x1.x,x1.y);
 printf("x2=%g+i(%g)\n", x2.x,x2.y);
}
```

10.21 Să se scrie o funcţie care adună două matrice pătratice de ordinul trei. Elementele matricelor sînt numere de tip *double*.

Funcţia de faţă şi cele care urmează în acest capitol vor folosi tipul MATP3 declarat astfel:

```
typedef struct {
   double x11,x12,x13,x21,x22,x23,x31,x32,x33;
} MATP3;
```

FUNCŢIA BX21

```
void admatp(MATP3 *a, MATP3 *b,MATP3 *c) /* c = a + b */
{
 c -> x11 = a -> x11 + b -> x11;
 c -> x12 = a -> x12 + b -> x12;
 c -> x13 = a -> x13 + b -> x13;
 c -> x21 = a -> x21 + b -> x21;
 c -> x22 = a -> x22 + b -> x22;
 c -> x23 = a -> x23 + b -> x23;
 c -> x31 = a -> x31 + b -> x31;
 c -> x32 = a -> x32 + b -> x32;
 c -> x33 = a -> x33 + b -> x33;
}
```

10.22 Să se scrie o funcţie care înmulţeşte la stînga o matrice pătratică de ordinul trei cu un vector de trei elemente.

Dacă matricea are elementele:

x11 x12 x13
x21 x22 x23
x31 x32 x33

iar vectorul este *b* de componente:

b[0],b[1],b[2]

atunci produsul dintre matrice şi vectorul *b* este un vector *c* de trei elemente:

c[i-1] = xi1*b[0] + xi2*b[1] + xi3*b[2] pentru i = 1,2,3.

FUNCŢIA BX22

```
void mulmatvect3(MATP3 *a, double b[],double c[])
/* c = a*b */
{
 c[0] = a->x11 * b[0] + a->x12 * b[1] +
        a->x13 * b[2];
 c[1] = a->x21 * b[0] + a->x22 * b[1] +
        a->x23 * b[2];
 c[2] = a->x31 * b[0] + a->x32 * b[1] +
        a->x33 * b[2];
}
```

10.23 Să se scrie o funcţie care înmulţeşte două matrice pătratice de ordinul trei.

FUNCŢIA BX23

```
void mulmatp3(MATP3 *a, MATP3 *b,MATP3 *c )
/* c = a * b */
{
 c -> x11 = a -> x11 * b -> x11 + a -> x12 *
      b -> x21 + a -> x13 * b -> x31;
 c -> x12 = a -> x11 * b -> x12 + a -> x12 *
      b -> x22 + a -> x13 * b -> x32;
 c -> x13 = a -> x11 * b -> x13 + a -> x12 *
      b -> x23 + a -> x13 * b -> x33;
 c -> x21 = a -> x21 * b -> x11 + a -> x22 *
```

```
        b -> x21 + a -> x23 * b -> x31;
  c -> x22 = a -> x21 * b -> x12 + a -> x22 *
        b -> x22 + a -> x23 * b -> x32;
  c -> x23 = a -> x21 * b -> x13 + a -> x22 *
        b -> x23 + a -> x23 * b -> x33;
  c -> x31 = a -> x31 * b -> x11 + a -> x32 *
        b -> x21 + a -> x33 * b -> x31;
  c -> x32 = a -> x31 * b -> x12 + a -> x32 *
        b -> x22 + a -> x33 * b -> x32;
  c -> x33 = a -> x31 * b -> x13 + a -> x32 *
        b -> x23 + a -> x33 * b -> x33;
}
```

10.24 Să se scrie o funcţie care negativează elementele unei matrice pătratice de ordinul 3.

FUNCŢIA BX24

```
void negmatp3(MATP3 *a, MATP3 *b) /* b = -a */
{
 b -> x11 = - a -> x11;
 b -> x12 = - a -> x12;
 b -> x13 = - a -> x13;
 b -> x21 = - a -> x21;
 b -> x22 = - a -> x22;
 b -> x23 = - a -> x23;
 b -> x31 = - a -> x31;
 b -> x32 = - a -> x32;
 b -> x33 = - a -> x33;
}
```

10.25 Să se scrie o funcţie care calculează şi returnează valoarea determinantului unei matrice pătratice de ordinul 3.

Calculul determinantului se face folosind regula lui *Saarus.*

FUNCŢIA BX25

```
double detmatp3(MATP3 *a)
/* calculeaza si returneaza valoarea determinantului matricei a */
{
 return a -> x11 * a -> x22 * a -> x33 +
        a -> x12 * a -> x23 * a -> x31 +
        a -> x21 * a -> x32 * a -> x13 -
```

```
        a -> x31 * a -> x22 * a -> x13 -
        a -> x11 * a -> x32 * a -> x23 -
        a -> x21 * a -> x12 * a -> x33;
}
```

10.26 Să se scrie o funcţie care calculează inversa unei matrice pătratice de ordinul 3. Funcţia returnează 0 dacă matricea are determinantul nul şi unu în caz contrar.

Fie *a* matricea de ordinul 3:

x11 x12 x13
x21 x22 x23
x31 x32 x33

Inversa matricei *a* este matricea:

X11 X21 X31
X12 X22 X32
X13 X23 X33

unde prin *Xij* am notat complementul algebric al elementului *xij* al matricei date. Dacă notăm cu *d* determinantul matricei *a*, atunci *Xij* se calculează astfel:

Xij = Aij / d   dacă i+j este par;

şi

Xij = -Aij / d   dacă i+j este impar,

unde prin *Aij* s-a notat determinantul matricei obţinut din matricea iniţială suprimînd linia a *i* - a şi coloana a *j* - a.

FUNCŢIA BX26

```
int invmatp3(MATP3 *a, MATP3 *b )
/* - in b se obtine inversa lui a;
   - returneaza:
         0 - daca determinantul matricei este 0;
         1 - in caz contrar. */
{
 double d;

 if(( d = detmatp3( a )) == 0 )
    return 0;
 b->x11=(a->x22 * a->x33 - a->x23 * a->x32)/d;
 b->x12=(a->x32 * a->x13 - a->x12 * a->x33)/d;
```

```
  b->x13=(a->x12 * a->x23 - a->x13 * a->x22)/d;
  b->x21=(a->x31 * a->x23 - a->x21 * a->x33)/d;
  b->x22=(a->x11 * a->x33 - a->x31 * a->x13)/d;
  b->x23=(a->x21 * a->x13 - a->x11 * a->x23)/d;
  b->x31=(a->x21 * a->x32 - a->x31 * a->x22)/d;
  b->x32=(a->x12 * a->x31 - a->x11 * a->x32)/d;
  b->x33=(a->x11 * a->x22 - a->x12 * a->x21)/d;
  return 1;
}
```

10.27 Să se scrie o funcţie care citeşte elementele unei matrice pătratice de ordinul 3 de tip *double*.

Funcţia returnează zero la intîlnirea sfîrşitului de fişier şi unu în caz contrar.

FUNCŢIA BX27

```
int citelem(double *x, int i,int j);   /* prototip */
int citmatp3(MATP3 *a)
/* - citeste elementele unei matrice patratice de ordinul 3;
   - returneaza:
          0 - la intilnirea sfirsitului de fisier;
          1 - in caz contrar. */
{
 double l;

 if(citelem(&i,1,1) == 0 )
     return 0;
 else
     a->x11 = l;
 if(citelem(&l,1,2) == 0 )
     return 0;
 else
     a->x12 = l;
 if(citelem(&l,1,3) == 0 )
     return 0;
 else
     a->x13 = l;
 if(citelem(&l,2,1) == 0)
     return 0;
 else
     a->x21 = l;
 if(citelem(&l,2,2) == 0 )
```

```
        return 0;
    else
        a->x22 = 1;
    if(citelem(&l,2,3) == 0 )
        return 0;
    else
        a->x23 = 1;
    if(citelem(&l,3,1) == 0 )
        return 0;
    else
        a->x31 = 1;
    if(citelem(&l,3,2) == 0 )
        return 0;
    else
        a->x32 = 1;
    if(citelem(&l,3,3) == 0 )
        return 0;
    else
        a->x33 = 1;
    return 1;
   } /* sfirsit functia citmatp3 */

   int citelem( double *x, int i,int j)
   /* - citeste un numar de tip double care reprezinta elementul unei
        matrice din linia i si coloana j;
      - returneaza:
            0 - la intilnirea sfirsitului de fisier;
            1 - altfel. */
   {
    char t[255];

    for( ; ; ) {
        printf("elementul din linia %d si\
                coloana %d: ",i,j );
        if(gets(t) == 0 )
            return 0;
        if(sscanf(t,"%lf",x) == 1 )
            return 1;
        printf("nu s-a tastat un numar\n");
    }
   }
```

10.28 Să se scrie un program care rezolvă un sistem de trei ecuaţii cu trei

necunoscute de forma:

(1) A*x = b

Programul de faţă rezolvă sistemul înmulţind relaţia (1) la stînga cu inversa lui *A*. Deci:

x = inv(A) * b

Paşii programului sînt:

1. Citeşte elementele matricei A

   (funcţia *citmatp3*, exerciţiul 10.27).

2. Citeşte termenul liber b

   (funcţia *pndcit*, exerciţiul 8.8).

3. Inversează matricea A

   (funcţia *invmatp3*, exerciţiul 10.26).

4. Calculează produsul la stînga dintre inversa matricei A şi termenul liber b

   (funcţia *mulmatvect3*, exerciţiul 10.22).

5. Afişează soluţia obţinută x0.

6. Afişează reziduurile, adică diferenţa:

   r = A * x0 - b.

   În acest scop se realizează:

   - produsul A*x0
     (funcţia *mulmatvect3*, exerciţiul 10.22);
   - diferenţa:
     A*x0 - b.

PROGRAMUL BX28

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
 double x11,x12,x13,x21,x22,x23,x31,x32,x33;
} MATP3;

#include "bviii8.cpp" /* pndcit */
```

```
#include "bx22.cpp" /* mulmatvect3 */
#include "bx25.cpp" /* detmatp3   */
#include "bx26.cpp" /* invmatp3   */
#include "bx27.cpp" /* citmatp3   */

main() /* - rezolva un sistem de 3 ecuatii cu 3 necunoscute inmultind
              la stinga cu inversa matricei sistemului;
            - afiseaza solutia determinata si reziduurile. */
{
 MATP3 a,inva;
 int i;
 double b[3],x0[3],r[3];
 char er[] = "s-a tastat EOF\n";

/* citeste matricea sistemului */
 if(citmatp3(&a) == 0 ) {
     printf(er);
     exit(1);
 }

/* citeste termenul liber */
 if(pndcit(3,b ) != 3 ) {
     printf(er);
     exit(1);
 }

/* inverseaza matricea sistemului */
 if(invmatp3(&a,&inva) == 0 ) {
     printf("determinantul sistemului este nul\n");
     exit(1);
 }

/* produsul la stinga dintre inversa matricei a si termenul liber b */
 mulmatvect3(&inva,b,x0);

/* afisarea solutiei */
 for( i=0; i < 3; i++)
     printf("x%d= %g\n", i+1,x0[i]);

/* calculeaza reziduurile */
/* calculeaza produsul a*x0 */
 mulmatvect3(&a,x0,r);
```

```
/* se afiseaza reziduurile */
 for( i = 0; i < 3; i++)
     printf("reziduu linia %d= %g\n",
                i+1, r[i] - b[i] );
}
```

Observaţii:

1. Toate funcţiile din exerciţiile 10.21 - 10.28, pot fi rescrise folosind tablouri de tip *double* în locul structurilor de tip MATP3. Propunem cititorului să facă acest lucru, funcţiile simplificîndu-se pe baza utilizării instrucţiunilor ciclice.

2. Funcţia *citelem* poate fi scrisă mai simplu dacă instrucţiunea:

```
if(citelem(&e,i,j) == 0 )
    return 0;
else
    a - > xij = e; pentru i,j = 1,2,3
```

se înlocuieşte cu:

```
if(citelem(&a -> xij,i,j) == 0)
    return 0;
```

10.29 Să se scrie o funcţie care calculează şi returnează cel mai mare divizor comun a două numere *m şi n* de tip *long*.

Funcţia utilizează algoritmul lui Euclid (vezi exerciţiul 4.24).

FUNCŢIA BX29

```
long cmmdc(long m, long n )/* calculeaza si returneaza (m,n) */
{
 long r;

 do {
     r=m%n;
     if(r) {
         m = n;
         n = r;
     }
 } while(r);
 return n;
}
```

Observaţie:

Funcţia presupune că n este diferit de zero.

10.30 Să se scrie o funcţie care calculează şi returnează cel mai mic multiplu comun a două numere *m şi n* de tip *long*.

Dacă notăm cu:

[m,n]

cel mai mic multiplu comun al numerelor *m şi n* , atunci:

[m,n] = m * n / (m,n)

unde prin:

(m,n) - Am notat cel mai mare divizor comun al aceloraşi numere.

FUNCŢIA BX30

```
long cmmmc(long m, long n) /* calculeaza si returneaza [m,n] */
{
 return m*n/cmmdc(m,n);
}
```

Observaţie:

Funcţia presupune că ambele numere sînt diferite de zero.

10.31 În acest exerciţiu şi în următoarele din acest capitol, vom utiliza tipul RATIONAL definit astfel:

```
typedef struct {
    long numarator;
    long numitor;
} RATIONAL;
```

O dată de tip RATIONAL o vom numi *fracţie*. Valoarea unei astfel de date este egală cu raportul:

numarator / numitor, dacă *numitor* este diferit de zero

şi

nu este definită, dacă *numitor* are valoarea zero

O fracţie este *ireductibilă* dacă numărătorul şi numitorul ei sînt numere

prime între ele. Pentru ca o fracţie să fie ireductibilă, este suficient să simplificăm fracţia respectivă cu cel mai mare divizor comun al numărătorului şi numitorului ei. Funcţia de faţă transformă o dată de tip RATIONAL într-o fracţie ireductibilă.

FUNCŢIA BX31

```
void simplifica(RATIONAL *a)
/* simplifica pe a cu (numarator,numitor) */
{
 long d;

 if(a -> numarator == 0 || a ->numitor == 0)
     return;
 d = cmmdc(a->numarator,a->numitor);
 a -> numarator /= d;
 a -> numitor /= d;
}
```

10.32 Să se scrie o funcţie care adună două date de tip RATIONAL.

Fie *a şi b* cele două date.

Adunarea se execută conform următorilor paşi:

1. Dacă numărătorul lui *a* este zero, rezultatul este *b* şi se revine din funcţie.
2. Dacă numărătorul lui *b* este zero, rezultatul este *a* şi se revine din funcţie.
3. Se calculează cel mai mic multiplu comun al numitorilor lui *a* şi *b*, fie acesta *m*.
4. Se calculează:

   a1 = (numărătorul lui a) * (m/(numitorul lui a)).
5. Se calculează *b1* ca şi *a1*, folosind numărătorul şi numitorul lui *b*.
6. Numărătorul rezultatului este suma:

   a1 + b1.
7. Numitorul rezultatului este *m*.
8. Se simplifică rezultatul dacă este posibil.

FUNCŢIA BX32

```
void adfr(RATIONAL *a, RATIONAL *b,RATIONAL *c)
/* c = a + b */
{
 long m;

 if(a -> numitor ==0 || b -> numitor ==0)
     return ;
 if(a->numarator == 0 ){
     c->numarator = b->numarator;
     c->numitor = b->numitor;
 }
 else
     if(b->numarator == 0) {
         c->numarator = a->numarator;
         c->numitor = a->numitor;
     }
     else {
         m=cmmmc(a->numitor,b->numitor);
         c->numarator = a->numarator*(m/a->numitor)
           + b->numarator * (m/b->numitor);
         c->numitor = m;
     }
 simplifica(c);
}
```

10.33 Să se scrie o funcţie care scade două date de tip RATIONAL.

FUNCŢIA BX33

```
void subfr(RATIONAL *a, RATIONAL *b,RATIONAL *c)
/* c = a-b */
{
 RATIONAL temp;
/* temp = -b */

 temp.numarator = -b->numarator;
 temp.numitor = b->numitor;
 adfr(a,&temp,c);
}
```

10.34 Să se scrie o funcţie care înmulţeşte două date de tip RATIONAL.

FUNCŢIA BX34

```
void mulfr(RATIONAL *a,RATIONAL *b,RATIONAL *c)
/* c = a*b */
{
 c->numarator = a->numarator * b->numarator;
 c->numitor = a->numitor * b->numitor;
 simplifica(c);
}
```

10.35 Să se scrie o funcţie care împarte două date de tip RATIONAL.

FUNCŢIA BX35

```
void divfr(RATIONAL *a,RATIONAL *b,RATIONAL *c)
/* c = a/b */
{
 c->numarator = a->numarator * b->numitor;
 c->numitor = a->numitor * b->numarator;
 simplifica(c);
}
```

10.36 Să se scrie o funcţie care citeşte numărătorul şi numitorul unei date de tip RATIONAL.

FUNCŢIA BX36

```
int citfr(RATIONAL *a)
/* - citeste numaratorul si numitorul unei fractii;
   - returneaza:
        0 - la intilnirea sfirsitului de fisier;
        1 - altfel. */
{
 char t[255];

 for ( ; ; ) {
     printf("numaratorul fractiei: ");
     if(gets(t) == 0 )
         return 0;
     if(sscanf(t,"%ld",&a->numarator) == 1)
         break;
     printf("nu s-a tastat un intreg\n");
 }
 for( ; ; ) {
```

```
    printf("numitorul fractiei: ");
    if(gets(t) == 0)
        return 0;
    if(sscanf(t,"%ld",&a->numitor) == 1 &&
                 a->numitor != 0 )
        break;
    printf("nu s-a tastat un intreg nenul\n");
 }
 return 1;
}
```

10.37 Să se scrie un program care citeşte componentele a trei date de tip RATIONAL, f1, f2 şi f3, calculează şi afişează sub formă de fracţie valoarea expresiei:

(f1 - f2) * f3

PROGRAMUL BX37

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
 long numarator;
 long numitor;
} RATIONAL ;

#include "bx29.cpp" /* cmmdc      */
#include "bx30.cpp" /* cmmmc      */
#include "bx31.cpp" /* simplifica */
#include "bx32.cpp" /* adfr       */
#include "bx33.cpp" /* subfr      */
#include "bx34.cpp" /* mulfr      */
#include "bx36.cpp" /* citfr      */

main() /* citeste datele f1,f2 si f3 de tip RATIONAL, calculeaza si
          afiseaza valoarea expresiei:
              (f1-f2)*f3
          sub forma de fractie. */
{
 RATIONAL f[3],temp;
 char er[] = "s-a tastat EOF\n";
 char *text[]={"prima fractie\n",
               "fractia a doua\n",
```

```
    "fractia a treia\n"
  };
  int i;

 /* citeste fractiile: f[0]=f1, f[1]=f2, f[2]=f3 */
  for(i=0; i<=2 ; i++) {
      printf(text[i]);
      if(citfr(&f[i]) == 0 ) {
          printf(er);
          exit(1);
      }
  }

 /* temp = f1 - f2 */
  subfr(&f[0],&f[1],&temp);

 /* temp =temp*f3 */
  mulfr(&temp,&f[2],&temp);

 /* scrie rezultatul */
  printf("(f1-f2)*f3:numarator= %ld\n",
         temp.numarator);
  printf("(f1-f2)*f3:numitor= %ld\n",temp.numitor);
 }
```

10.4. Reuniune

Limbajul C oferă utilizatorului posibilitatea de a păstra într-o zonă de memorie date de tipuri diferite. Pînă în prezent am văzut că unei date i se alocă o zonă de memorie potrivit tipului datei respective şi în zona alocată ei se pot păstra numai date de acel tip. De exemplu, dacă avem în vedere declaraţia:

long x;

atunci pentru *x* se alocă 32 de biţi şi în zona respectivă se păstrează întregi reprezentaţi prin complement faţă de 2.

Se pot ivi situaţii în care în momente diferite ale execuţiei, am dori ca în aceeaşi zonă de memorie să putem păstra date de tipuri diferite. De exemplu, dacă după un timp nu mai este nevoie de variabila *x*, zona alocată lui *x* ar putea fi utilizată în alte scopuri, pentru a păstra o dată de un alt tip: *char, int* sau chiar *float*. Astfel de *reutilizări* ale zonelor de memorie pot conduce la economisirea memoriei.

Aceasta este posibil "grupînd" împreună datele care dorim să fie alocate în aceeaşi zonă de memorie. Pentru a realiza o astfel de grupare se foloseşte o construcţie similară cu cea pentru *structuri*, diferenţa constînd în aceea că se schimbă cuvîntul *struct* cu *union*. În rest, toate formatele întîlnite în cazul structurilor rămîn valabile. Tipul introdus prin *union* este un tip *definit* de utilizator ca şi cel definit prin *struct*. O astfel de dată grupată o vom numi *reuniune*.

Exemple:

1.

```
union a {
 int x;
 long y;
 double r;
 char c;
} p;
```

p este o reuniune de tipul *a*.

Componentele *x,y,r şi c* ale lui *p* le referim ca şi în cazul structurilor prin:

p.x, p.y, p.r şi p.c

Ele sînt alocate în aceeaşi zonă de memorie şi de aceea, la un moment dat al execuţiei, numai una din aceste componente este definită(alocată).

Pentru *p* se alocă o zonă de memorie suficientă pentru a păstra data care necesită numărul maxim de octeţi, deci se vor aloca 8 octeţi necesari pentru a putea păstra componenta *r* de tip *double*.

Să observăm că dacă înlocuim cuvîntul *union* prin *struct*, atunci pentru *p* se alocă:

- 2 octeţi pentru *x*;
- 4 octeţi pentru *y*;
- 8 octeţi pentru *r*;
- 1 octet pentru *c*,

deci în total 15 octeţi. Aceasta deoarece, în cazul unei structuri, componentele ei sînt definite simultan şi deci trebuie să fie alocate în zone diferite de memorie.

2.

```
typedef union {
  char nume[70];
```

```
 int nrmat;
 long cod;
} ZC;

ZC sir;
```

sir este o reuniune de tip *ZC* pentru care s-au alocat 70 de octeţi. În această zonă se pot păstra şiruri de caractere la care ne putem referi prin:

sir.nume;
sir.nume[0];
sir.nume[1];
etc.

sau întregi de tip *int* sau *long*. La aceştia ne referim prin:

sir.nrmat

şi

sir.cod.

Fie:

ZC *p;

atunci putem realiza o atribuire de forma:

p = &sir;

Prin intermediul lui *p* ne putem referi la componentele reuniunii *sir* astfel:

p -> nume;
p -> nume[0];
p -> nume[1]
etc.
p -> nrmat

şi

p -> cod.

În legătură cu reuniunile, pot apărea probleme la utilizarea lor, deoarece programatorul trebuie să cunoască, în fiecare moment al execuţiei, ce componentă a reuniunii este prezentă în zona alocată ei.

Fie reuniunea:

```
union {
 int i;
 float f;
```

```
    double d;
} zc;
```

Dacă la un moment dat, în zona alocată datei *zc* se păstrează un întreg de tip *int* (de exemplu: se face atribuirea zc.i = 10) şi se fac referiri la componenta *d*:

if(zc.d > 0)

atunci rezultatul comparării va fi eronat. Această eroare nu poate fi semnalată de compilator şi nici la execuţie şi de aceea, astfel de utilizări pot fi evitate numai de programator.

Pentru a înlătura erorile de acest fel, se recomandă utilizarea unui indicator care să definească tipul datei păstrate în fiecare moment în zona alocată reuniunii respective. De exemplu, în cazul reuniunii *zc* de mai sus este necesar un indicator care să aibă trei valori corespunzătoare celor trei tipuri ale componentelor lui *zc (int, float* şi *double*). Aceste valori le numim sugestiv prin constante simbolice:

```
#define INTREG 1
#define FSIMPLU 2
#define FDUBLU 3
```

Adăugăm la reuniunea de mai sus un indicator care are una din aceste valori, în funcţie de componenta prezentă în zona reuniunii. Se obţine tipul utilizator de mai jos:

```
struct tszc {
 int tipcrt; /* indicator care defineste tipul curent */
 union {
     int i;
     float f;
     double d;
 } zc;
} ;
```

Declarăm structura de tip *tszc:*

struct tszc szc;

Structura are două componente:

tipcrt - Este de tip int.

zc - Este o reuniune de componente *i,f şi d.*

Deci, tot timpul este alocată data *tipcrt* şi numai una din componentele *i,f,d.*

Pentru a păstra o dată de tip *int*, de exemplu valoarea 123, vom folosi atribuirea:

```
szc.zc.i = 123;
```

Alături de această atribuire vom mai utiliza încă una şi anume:

```
szc.tipcrt = INTREG;
```

În felul acesta, componenta *tipcrt* ne permite să stabilim faptul că reuniunea *zc* conţine o dată de tip *int*.

În mod analog, dacă dorim să păstrăm, în zona respectivă, o dată flotantă de tip *double*, de exemplu valoarea lui *pi*, vom folosi atribuirile:

```
szc.zc.d = 3.14159265;
szc.tipcrt = FDUBLU;
```

La utilizarea datelor păstrate în acest fel se poate folosi o construcţie *if* de forma:

```
if(szc.tipcrt == INTREG)
   se foloseşte szc.zc.i
else if(szc.tipcrt == FSIMPLU)
      se foloseşte szc.zc.f
   else if(szc.tipcrt == FDUBLU)
         se foloseşte szc.zc.d
      else
         eroare
```

Aceeaşi utilizare se obţine folosind o instrucţiune *switch:*

```
switch (szc.tipcrt) {
   case INTREG:
      se foloseşte szc.zc.i
      break;
   case FSIMPLU:
      se foloseşte szc.zc.f
      break;
   case FDUBLU:
      se foloseşte szc.zc.d
      break;
   default:
      eroare
}
```

În încheierea acestui paragraf amintim că reuniunile *nu pot* fi iniţializate, spre deosebire de structuri.

Exerciţii:

10.38 Fie tipul FIG declarat ca mai jos:

```
typedef struct {
    int tip; /* tipul figurii */
    union {
        double raza;   /* cerc       */
        double lp;     /* patrat     */
        double ld[2];  /* dreptunghi */
        double lt[3];  /* triunghi   */
    } fig;
} FIG;
```

O dată de tip FIG conţine elementele unei figuri necesare pentru a calcula aria figurii respective. Figurile avute în vedere sînt:

- cerc;
- pătrat;
- dreptunghi

şi

- triunghi.

În cazul primelor două figuri, data de tip FIG conţine o valoare de tip *double* care, în cazul cercului reprezintă lungimea razei acestuia, iar în cazul pătratului, lungimea laturii pătratului. În cazul dreptunghiului, data conţine două elemente de tip *double:* lungimea şi lăţimea. În sfîrşit, în cazul triunghiului, data conţine trei valori de tip *double* care reprezintă lungimile celor trei laturi ale triunghiului.

Componenta *tip* defineşte elementele(figura) prezente într-o dată de tip FIG şi are valorile:

0 - Pentru cerc.

1 - Pentru pătrat.

2 - Pentru dreptunghi.

3 - Pentru triunghi.

-1 - Pentru eroare.

În locul acestor valori, considerăm constantele simbolice:

```
#define EROARE -1
#define CERC 0
#define PATRAT 1
#define DREPTUNGHI 2
```

```
#define TRIUNGHI 3
```

Funcţia de mai jos are ca parametru o dată de tip FIG, calculează şi returnează aria figurii ale cărei elemente sînt conţinute în zona definită de parametru.

Funcţia returnează 0 în cazul în care datele sînt eronate.

La calculul ariei unui triunghi se foloseşte formula lui Heron.

FUNCŢIA BX38

```
#define PI 3.14159265358979

double aria(FIG *p)
/* calculeaza si returneaza aria figurii definite de elementele prezente in
   zona spre care pointeaza p; la eroare returneaza 0 */
{
 double sp,a,b,c;

 switch(p->tip) {
     case CERC:
         return PI*p->fig.raza*p->fig.raza;
     case PATRAT:
         return p->fig.lp*p->fig.lp;
     case DREPTUNGHI:
         return p->fig.ld[0]*p->fig.ld[1];
     case TRIUNGHI:
         sp=(p->fig.lt[0] + p->fig.lt[1] +
            p->fig.lt[2])/2;
         if((a=sp - p->fig.lt[0]) > 0 &&
           (b=sp - p->fig.lt[1]) > 0 &&
           (c=sp - p->fig.lt[2]) > 0 )
                return sqrt(sp*a*b*c);
         else { /* cele 3 valori nu reprezinta lungimile laturilor
                   unui triunghi */
             printf("a= %g\tb= %g\tc= %g\tnu\
                   formeaza un triunghi\n",
                   p->fig.lt[0], p->fig.lt[1],
                   p->fig.lt[2] );
             return 0;
         }
     default:/* eroare */
         return 0;
```

```
    }
  }
```

10.39 Să se scrie o funcţie care are ca parametru un pointer spre o dată de tip FIG şi care citeşte şi păstrează elementele figurii definite de componenta *tip* a datei de tip FIG.

FUNCŢIA BX39

```
int citfig(FIG *p)
/* - citeste elementele figurii definite de p->tip;
   - returneaza:
      0 - la intilnirea sfirsitului de fisier sau la eroare;
      1 - altfel. */
{
 char t[255];

 switch(p->tip) {
    case CERC:
        for( ; ; ) {
            printf("raza= ");
            if(gets(t) == 0 )
               return 0;
            if(sscanf(t,"%lf",&p->fig.raza)== 1 &&
                  p->fig.raza>0)
               return 1;
            printf("nu s-a tastat un numar\
                    pozitiv\n");
        }
    case PATRAT:
        for( ; ; ) {
            printf("latura patratului= ");
            if(gets(t) == 0 )
               return 0;
            if(sscanf(t,"%lf",&p->fig.lp) == 1 &&
                 p->fig.lp > 0)
               return 1;
            printf("nu s-a tastat un numar\
                    pozitiv\n");
        }
    case DREPTUNGHI:
        for( ; ; ) {
            printf("lungimea si latimea pe aceeasi\
```

```
                linie: ");
            if(gets(t) == 0 )
                return 0;
            if(sscanf(t,"%lf %lf",&p->fig.ld[0],
                   &p->fig.ld[1])==2 &&
                    p->fig.ld[0] > 0 &&
                    p->fig.ld[1] > 0)
                return 1;
            printf("nu s-au tastat 2 numere\
                    pozitive\n");
        }
    case TRIUNGHI:
        for( ; ; ) {
            printf("laturile triunghiului pe\
                    aceeasi linie: ");
            if(gets(t) == 0 )
                return 0;
            if(sscanf(t,"%lf %lf %lf",&p->fig.lt[0],
                  &p->fig.lt[1],
                  &p->fig.lt[2])==3 &&
                   p->fig.lt[0]>0 &&
                   p->fig.lt[1]>0 &&
                   p->fig.lt[2]> 0 )
                return 1;
            printf("nu s-au tastat 3 numere\
                    pozitive\n");
        }
    default:
        return 0;
  }
}
```

10.40 Să se scrie un program care citeşte elementele unei figuri necesare pentru calculul ariei sale. Elementele sînt precedate de o literă mare, care defineşte figura. Această literă se tastează la început, singură pe o linie.

Programul afişează aria figurii.

Corespondenţa dintre litere şi figuri este următoarea:

C - Pentru cerc.
P - Pentru pătrat.
D - Pentru dreptunghi.

T - Pentru triunghi.

PROGRAMUL BX40

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

typedef struct {
 int tip;
 union {
    double raza;
    double lp;
    double ld[2];
    double lt[3];
 } fig;
} FIG;

#define EROARE -1
#define CERC 0
#define PATRAT 1
#define DREPTUNGHI 2
#define TRIUNGHI 3

#include "bx38.cpp" /* arie */
#include "bx39.cpp" /* citfig */

main() /* - citeste o litera mare care defineste o figura geometrica,
            apoi citeste elementele figurii respective;
          - calculeaza si afiseaza aria acelei figuri. */
{
 char t[255];
 char er[]="s-a tastat EOF\n";
 char lit[2];
 FIG f;
 double a;

 for( ; ; ) {
    printf("tastati una din literele mari:\n");
    printf("C\nD\nP\nT\n");
    if(gets(t) == 0 ){
       printf(er);
       exit(1);
```

```
        }
        sscanf(t,"%ls",lit);
        switch(lit[0]) {
            case 'C': /* cerc */
                f.tip = CERC;
                break;
            case 'D': /* dreptunghi */
                f.tip = DREPTUNGHI;
                break;
            case 'P': /* patrat */
                f.tip = PATRAT;
                break;
            case 'T': /* triunghi */
                f.tip = TRIUNGHI;
                break;
            default: /* eroare */
                printf("nu s-a tastat una din literele\
                        mari C,D,P sau T\n");
                f.tip = EROARE;
        }
        if(f.tip != EROARE)
            break;
    } /* sfirsit for */

  /* citeste elementele figurii */
    if(citfig(&f) == 0 ) {
        printf(er);
        exit(1);
    }

  /* calculeaza si afiseaza aria figurii */
    if((a=aria(&f)) == 0 )
        exit(1);
    printf("aria figurii= %g\n", a );
  }
```

10.41 Fie tipul utilizator introdus prin următoarea declarație:

```
typedef struct d_c {
    int zz;
    int tipluna;
    union {
        int nll;
```

```
            char sluna[11];
        } luna;
        int an;
    } D_C ;
```

unde:

tipluna - Poate avea una din valorile:

INTREG

sau

SIR

Dacă *tipluna* are valoarea INTREG, atunci luna se exprimă prin numărul ei care se păstrează ca valoare a variabilei *nll*. În caz contrar, luna este păstrată prin denumirea ei în tabloul *sluna*.

Funcţia de faţă are ca parametru un pointer spre o dată de tip *D_C* şi returnează numărul lunii calendaristice. La eroare, returnează valoarea 0.

În cazul în care luna calendaristică se dă prin denumirea ei, se utilizează tabloul de pointeri *tpdl* definit în exerciţiul 8.19.

FUNCŢIA BX41

```
int nrluna(D_C *p) /* returneaza numarul lunii sau 0 la eroare */
{
 int i;
 static char *tpdl[]= {"",
     "ianuarie",
     "februarie",
     "martie",
     "aprilie",
     "mai",
     "iunie",
     "iulie",
     "august",
     "septembrie",
     "octombrie",
     "noiembrie",
     "decembrie"
 };

 if(p->tipluna == INTREG)
/* luna se da prin numarul ei */
     return p->luna.nll <1 ||
```

```
        p->luna.nll >12 ? 0 : p->luna.nll;

    if(p->tipluna != SIR )
   /* tipluna nu are o valoare corecta */
        return 0;

   /* se cauta denumirea lunii in tabloul tpdl */
    for( i=1; i <13; i++)
        if(strcmp(p->luna.sluna,tpdl[i]) == 0 )
            return i;
    return 0;
   }
```

10.42 Funcţia din acest exerciţiu are ca parametru un pointer spre o dată de tip *D_C*, validează data calendaristică definită de pointerul respectiv şi returnează numărul lunii din data respectivă. În cazul în care data este eronată, funcţia returnează valoarea zero.

Funcţia validează data calendaristică apelînd funcţia *v_calend* definită în exerciţiul 6.5.

FUNCŢIA BX42

```
   int vnrluna(D_C *p)
   /* - valideaza data calendaristica spre care pointeaza p;
     - returneaza numarul lunii calendaristice si 0 la eroare. */
   {
    int nr;

    if((nr=nrluna(p)) == 0)
        return 0; /* luna eronata */
    if(v_calend(p->zz,nr,p->an))
        return nr; /* data calendaristica valida */
    return 0; /* data calendaristica eronata */
   }
```

10.43 Să se scrie un program care afişează diferenţa, în zile, dintre două date calendaristice. Una dintre cele două date se dă sub formă de argument în linia de comandă, iar cealaltă se citeşte de la tastatură.

Notăm cu *d1* data din linia de comandă şi cu *d2* data care se tastează. Programul afişează diferenţa *d2 - d1,* în valoare absolută.

Data *d1* are formatul:

zi luna an

unde:

zi	- Este un întreg de 1-2 cifre.
luna	- Este denumirea lunii calendaristice.
an	- Este întreg de 4 cifre şi reprezintă anul din intervalul [1600,4900].

Data *d2* are un format similar cu deosebirea că luna calendaristică se dă prin numărul ei.

Se pot tasta mai multe date *d2* la o aceeaşi execuţie a programului.

PROGRAMUL BX43

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct {
 int zz;
 int tipluna;
 union {
    int nll;
    char sluna[11];
 } luna;
 int an;
} D_C;

#define INTREG 1
#define SIR 2
#define EROARE -1

#include "bvi5.cpp"    /*v_calend   */
#include "bvi6.cpp"    /*zi_din_an  */
#include "bx41.cpp"    /* nrluna    */
#include "bx42.cpp"    /* vnrluna   */
#include "bviii2.cpp"  /* pcit_int  */
#include "bviii3.cpp"  /*pcit_int_lim */

int nrzile[] = { 0,31,28,31,30,31,30,
                 31,31,30,31,30,31};
```

```
main(int argc, char *argv[])
/* afiseaza numarul de zile dintre doua date calendaristice valide */
{
 char er[]="s-a tastat EOF\n";
 D_C d1,d2;
 int n,n1,n2,t,min,max;

/* pastreaza data calendaristica din linia de comanda in structura d1 */
 if(argc != 4) {
    printf("numar argumente= %d este eronat\n",
           argc);
    exit(1);
 }
 if(sscanf(argv[1],"%d",&d1.zz) != 1) {
    printf("ziua din linia de comanda eronata\n");
    exit(1);
 }
 if(sscanf(argv[2],"%10s",d1.luna.sluna) != 1) {
    printf("luna din linia de comanda eronata\n");
    exit(1);
 }
 if(sscanf(argv[3],"%d",&d1.an) != 1) {
    printf("anul din linia de comanda eronat\n");
    exit(1);
 }
 d1.tipluna = SIR;

/* valideaza data din d1 */
 if((n=vnrluna(&d1)) == 0 ) {
    printf("data calendaristica din linia de\
          comanda eronata\n");
    printf("zi: %d\tluna: %s\tanul: %d\n",d1.zz,
          d1.luna.sluna, d1.an);
    exit(1);
 }

/* numar zile din an */
 n1=zi_din_an(d1.zz,n,d1.an);

 for( ; ; ) {
/* citeste o data calendaristica de la tastatura si o pastreaza
  in structura d2 */
    if(pcit_int_lim("ziua ",1,31,&d2.zz) == 0)
```

```
        exit(0); /* s-a tastat EOF */
    if(pcit_int_lim("numarul lunii calendaristice"
             ,1,12, &d2.luna.nll) == 0 ) {
        printf(er);
        exit(1);
    }
    if(pcit_int_lim("anul ",1600,4900,&d2.an)==0 )
    {
        printf(er);
        exit(1);
    }
    d2.tipluna = INTREG;

/* valideaza data calendaristica din d2 */
    if(vnrluna(&d2) == 0) {
        printf("data calendaristica tastata este\
              eronata\n");
        printf("zi: %d\tluna: %d\tanul: %d\n",
             d2.zz,d2.luna.nll,d2.an);
        continue;
    }

/* numar zile din an */
    n2=zi_din_an(d2.zz,d2.luna.nll,d2.an);

/* determina relatia dintre ani */
    if(d1.an < d2.an) {
        min=d1.an;
        max=d2.an;
    }
    else {
        min = d2.an;
        max = d1.an;
    }

/* insumeaza diferenta dintre ani in zile */
    for(t=0;min < max; min++)
        t += 365 +(min%4==0 && min%100 ||
                   min%400 == 0);
    if(min == d1.an)
        t += n1 - n2; /* t=d1-d2 */
    else
        t += n2- n1; /* t = d2 - d1 */
```

```
        if(t < 0 )
            t = -t;

        printf("dl= %d/%d/%d\td2= %d/%d/%d\n",dl.zz,
                    n,dl.an,d2.zz,d2.luna.nll,d2.an);
        printf("abs(d2 - dl) = %d\n",t);
    }
}
```

10.5. Cîmp

Limbajul C permite definirea şi prelucrarea datelor pe *biţi*. Utilizarea lor poate conduce la economisirea de memorie. Într-adevăr, adesea avem nevoie de date care au numai două valori, zero sau unu. O astfel de dată poate fi păstrată pe un singur bit. De aceea, pentru astfel de date, nu se justifică să alocăm un octet sau chiar doi. În general, nu este util ca date de valori mici să fie păstrate pe octeţi sau pe 16 biţi, mai ales atunci cînd aceste date sînt în număr mare. În acest scop, limbajul C oferă posibilitatea de a declara date care să se *aloce pe biţi*.

Un şir de biţi adiacenţi formează un *cîmp*. Un cîmp trebuie să se poată păstra într-un cuvînt calculator.

Mai multe cîmpuri pot fi păstrate într-un acelaşi cuvînt calculator.

Cîmpurile se grupează formînd o structură. O astfel de structură se declară ca o structură obişnuită care are ca şi componente cîmpuri:

struct *nume* {
  cimp_1;
  cimp_2;
  ...
  cimp_n;
} *nume1,nume2,...,numem;*

Un cîmp se declară astfel:

tip nume_cimp: *lungime_in_biţi*

sau

tip: *lungime_in_biţi*

De obicei, *tip* este cuvîntul cheie *unsigned*, ceea ce înseamnă că şirul de biţi din cîmpul respectiv se interpretează ca fiind un întreg fără semn. Alte posibilităţi pentru *tip* sînt:

- int;
- unsigned char;

şi

- char.

Cîmpurile se alocă de la biţii de ordin inferior ai cuvîntului spre cei de ordin superior.

Cîmpurile cu semn se utilizează pentru a păstra întregi de valori mici prin complement faţă de doi. De aceea, în acest caz, bitul cel mai semnificativ al cîmpului este bit semn.

Dacă un cîmp nu se poate aloca în cuvîntul curent, el se va aloca în cuvîntul următor.

Un cîmp fără *nume* nu se poate referi. El defineşte o zonă neutilizată dintr-un cuvînt.

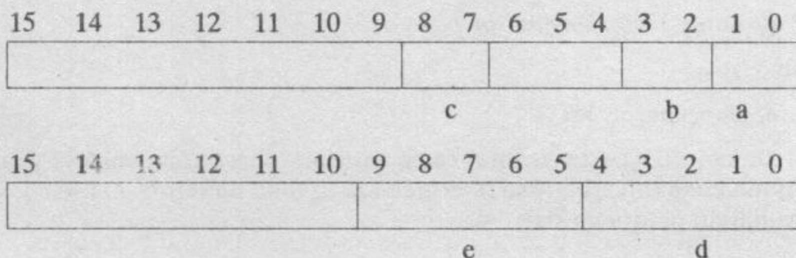
Lungimea în biţi poate fi egală cu zero. În acest caz, data următoare se alocă în cuvîntul următor.

Cîmpurile se pot referi folosind aceleaşi convenţii ca şi în cazul structurilor obişnuite.

Exemplu:

```
struct {
 unsigned a:2;
 int      b:2;
 unsigned  :3;
 unsigned c:2;
 unsigned  :0;
 int      d:5;
 unsigned e:5;
} x,y;
```

Pentru *x* se alocă două cuvinte, astfel:



x.a = 1 - Atribuie cîmpului *a* al datei *x* valoarea 1, deci bitul 0 devine 1, iar bitul 1 ia valoarea 0.

x.b = -1 - Atribuie cîmpului *b*, al datei *x*, valoarea -1.

- Aceasta înseamnă că ambii biţi ai lui *b* se fac egali cu 1 (11 este reprezentarea lui -1 pe 2 biţi prin complement faţă de 2).

Nu se pot defini tablouri de cîmpuri. De asemenea, operatorul adresă (& unar) nu se poate aplica la un cîmp.

Datele pe biţi conduc la programe care, de obicei, nu sînt portabile sau au o portabilitate redusă. De aceea, se recomandă utilizarea lor cu precauţie.

De asemenea, datele pe biţi necesită instrucţiuni suplimentare (deplasări, setări şi/sau mascări de biţi etc.)faţă de cazul cînd sînt păstrate în mod obişnuit (ca date de tip *int* sau *char*). De aceea, utilizarea lor se justifică numai atunci cînd alocarea pe biţi conduce la o economie substanţială de memorie faţă de alocarea pe octeţi sau pe cuvinte de 16 biţi.

Observaţie:

Prelucrarea datelor pe biţi se poate realiza şi fără a defini cîmpuri de biţi, utilizînd operatorii logici pe biţi. Utilizarea lor poate conduce însă la un efort de programare suplimentar care poate fi destul de mare. De asemenea, utilizarea operatorilor respectivi poate să nu fie făcută optim sau să conducă la folosirea unor expresii eronate.

Aceasta nu înseamnă că trebuie să renunţăm la utilizarea operatorilor logici pe biţi. Există situaţii cînd utilizarea lor permite scrierea unor programe mai performante decît dacă se utilizează, în aceleaşi scopuri, cîmpuri de biţi.

Exerciţii:

10.44 Să se scrie o funcţie care stabileşte dacă o relaţie binară este o relaţie de echivalenţă.

Fie A şi B două mulţimi (disjuncte sau nu, care pot şi coincide).

Prin *AxB* se notează mulţimea perechilor ordonate de forma:

<a,b>

unde:

a - Este un element al mulţimii A, iar *b* un element al mulţimii B.

Mulţimea *AxB* se numeşte *produsul cartezian* al mulţimilor A şi B.

O submulţime a produsului cartezian *AxB* se numeşte *relaţie binară*.

Dacă A = B, atunci o submulţime a produsului cartezian AxA defineşte o relaţie binară pe mulţimea A.

Fie R o relaţie binară, adică o submulţime a produsului cartezian AxB. Dacă perechea:

<a,b>

aparţine lui R, atunci se spune că *a şi b* se află în relaţia R. Notăm acest lucru prin:

(1) aRb

şi se obişnuieşte să se spună că expresia (1) este *adevărată*. Dacă *a şi b* nu sînt în relaţia R, atunci spunem că expresia (1) este *falsă*.

Exemple:

1. Dacă A = B = N, unde prin N am notat mulţimea numerelor naturale, atunci sînt definite o serie de relaţii binare pe mulţimea N. De exemplu, relaţia *mai mic* este definită pe mulţimea N şi ea se notează prin simbolul <.

   De exemplu, perechea:

   <3,7>

   aparţine relaţiei mai mic (<) şi aceasta înseamnă că expresia:

   3 < 7

   este adevărată. În schimb expresia:

   7 < 3

   este falsă.

2. Fie mulţimile:

   A = {1,2,3} şi B = { 2,5,7,8}

   Atunci:

   AxB = {<1,2>, <1,5>, <1,7>, <1,8>,<2,2>, <2,5>, <2,7>, <2,8>, <3,2>, <3,5>, <3,7>, <3,8> }

   Relaţia R este submulţimea perechilor produsului cartezian cu proprietatea că primul element divide pe al doilea.

   Atunci:

   R = { <1,2>, <1,5>, <1,7>, <1,8>,<2,2>, <2,8> }

Dacă notăm cu R1 submulţimea perechilor produsului *AxB* cu proprietatea că primul element este multiplul celui de-al doilea, atunci relaţia R1 este o relaţie vidă.

Fie R o relaţie definită pe mulţimea A.

Dacă

aRa

este o expresie adevărată pentru orice *a* din *A*, atunci se spune că relaţia R este *reflexivă*.

Relaţia < definită pe mulţimea numerelor naturale nu este o relaţie reflexivă. Pe aceeaşi mulţime se poate defini relaţia de egalitate = .

Perechea:

<a,a>

este în relaţia =, oricare ar fi *a* un număr natural, deci relaţia = este reflexivă.

Relaţia R definită pe mulţimea A se spune că este *simetrică* dacă din faptul că expresia:

aRb

este adevărată, rezultă că şi expresia:

bRa

este adevărată.

Relaţia <, definită pe mulţimea numerelor naturale nu este simetrică. În schimb, relaţia = este simetrică.

În sfirşit, relaţia R este *tranzitivă* dacă din faptul că expresiile:

aRb şi bRc

sînt adevărate, rezultă că şi expresia:

aRc

este adevărată.

Relaţiile < şi = definite pe mulţimea numerelor naturale N sînt tranzitive.

Dacă expresiile:

a < b şi b < c

sînt adevărate, rezultă că şi expresia:

a < c

este adevărată oricare ar fi *a,b,c* numere naturale.

O relaţie R definită pe mulţimea A, care este reflexivă, simetrică şi tranzitivă, se numeşte relaţie de *echivalenţă.*

Relaţia = este un exemplu simplu de relaţie de echivalenţă.

Relaţia < nu este o relaţie de echivalenţă nefiind simetrică şi nici reflexivă.

Relaţia <= (mai mic egal) este reflexivă dar nu şi simetrică, deci nu este o relaţie de echivalenţă.

Relaţia != (diferit) este o relaţie simetrică dar nu este reflexivă.

Un exemplu nebanal de relaţie de echivalenţă este relaţia de asemănare definită pe mulţimea triunghiurilor plane. Această relaţie se notează cu simbolul ~.

Dacă *a şi b* sînt două triunghiuri plane asemenea, atunci din faptul că *a* este asemenea cu *b*, rezultă că şi *b* este asemenea cu *a*. Deci, relaţia de asemănare este simetrică. Ea este şi reflexivă, deoarece un triunghi este totdeauna asemenea cu el însuşi.

În sfîrşit, dacă *a,b,c* sînt trei triunghiuri şi au loc relaţiile:

a ~ b şi b ~ c,

atunci are loc şi relaţia:

a ~ c.

În acest exemplu, precum şi în continuare, vom avea în vedere numai relaţii cu un număr finit de elemente. Relaţiile de acest fel se pot reprezenta prin matrici.

Dacă relaţia R este o submulţime a produsului cartezian AxB, atunci reprezentăm relaţia respectivă prin matricea *a* astfel:

- Fiecărui element din mulţimea A i se pune în corespondenţă o linie în matricea *a*.
- Fiecărui element din mulţimea B i se pune în corespondenţă o coloană în matricea *a*.
- Fie *ai* elementul mulţimii A căruia îi corespunde linia a i-a din matricea *a* şi *bj* elementul mulţimii B căruia îi corespunde coloana a j-a din matricea *a*. Atunci:

  a[i][j] = 1, dacă ai R bj

  şi

a[i][j] = 0, altfel.

Exemplu:

Să considerăm exemplul 2 de mai sus. Matricea *a* conţine 3 linii şi 4 coloane. Relaţia R se reprezintă prin următoarea matrice:

		2	5	7	8
	1	1	1	1	1
A	2	1	0	0	1
	3	0	0	0	0
			B		

În cazul în care A = B, matricea este pătratică. Relaţiile de echivalenţă se definesc pentru astfel de cazuri. Proprietatea de reflexivitate se exprimă simplu şi anume, matricea conţine 1 pe diagonala principală.

Într-adevăr, relaţia R este reflexivă dacă:

ai R ai

este adevărată, adică dacă

a[i][i] = 1, pentru toate valorile lui i.

Relaţia R este simetrică dacă matricea de reprezentare a ei este simetrică.

Într-adevăr, dacă R este simetrică, atunci din faptul că expresia:

ai R aj

este adevărată, rezultă că şi expresia:

aj R ai

este adevărată. Aceasta înseamnă că:

a[i][j] = a[j][i] = 1

iar dacă:

a[i][j] = 0, atunci a[j][i] = 0

deci:

a[i][j] = a[j][i]

Pentru a verifica tranzitivitatea relaţiei, să considerăm definiţia tranzitivităţii:

R este tranzitivă dacă din faptul că expresiile:

ai R aj şi aj R ak

sînt adevărate, rezultă că şi expresia:

ai R ak

este adevărată.

Deci, este necesar să se verifice că dacă

a[i][j] = a[j][k] = 1

atunci şi

a[i][k] = 1, pentru toate valorile lui i, j şi k.

Dacă există cel puţin un element

a[i][k] = 0

în timp ce

a[i][j] = a[j][k] = 1

pentru cel puţin un j, atunci relaţia nu este tranzitivă.

Elementele matricei de reprezentare a unei relaţii au numai două valori: 0 sau 1.

De aceea, o astfel de matrice se poate păstra pe biţi. Aceasta permite realizarea unei economii de memorie importantă faţă de cazurile în care matricea se păstrează pe octeţi sau cuvinte de 16 biţi. De exemplu, dacă se utilizează o matrice de ordinul 100, atunci necesarul de memorie va fi:

100*100 biţi = 10000/8 octeţi= 1250 octeţi,

faţă de 10000 de octeţi, cînd matricea s-ar păstra pe octeţi.

Avînd în vedere acest fapt, funcţia de faţă utilizează o matrice cu elemente păstrate pe biţi, pentru a reprezenta relaţia de analizat.

Ea returnează valoarea 1 dacă relaţia este de echivalenţă şi zero în caz contrar.

Matricea relaţiei se păstrează într-un tablou unidimensional de tip *char* şi fiecare element al tabloului păstrează 8 elemente consecutive din matricea respectivă.

Fie *a* matricea relaţiei, care este o matrice pătratică de ordinul *n*.

Notăm cu *ta* tabloul unidimensional care păstrează elementele matricei *a* pe biţi. Problema care ne interesează este accesul la elementul a[i][j] al matricei *a*, reprezentată prin tabloul *ta* (i = 0,1,...,n-1 şi j = 0,1,...,n-1).

Dacă elementele matricei *a* s-ar păstra pe octeţi, atunci folosind relaţia de liniarizare (vezi exerciţiul 5.1), acest element ar avea indicele:

k = i*n + j

deci

a[i][j] ar fi elementul ta[k = i*n+j].

Deoarece fiecare element al tabloului *ta* păstrează 8 elemente din matricea *a*, rezultă că valoarea lui *k*, de mai sus, trebuie micşorată de 8 ori.

Avem:

a[0][0], a[0][1],...,a[0][7], se păstrează în ta[0],
a[0][8], a[0][9],...,a[0][15], se păstrează în ta[1]
...

În general, elementul a[i][j] se păstrează în octetul de indice (i*n+j)/8, adică este un bit al elementului:

ta[(i*n+j)/8]

Poziţia bitului se determină astfel:

- dacă
  (i*n+j)%8 este zero,
  atunci este bitul cel mai semnificativ (bitul 7)
- dacă
  (i*n+j)%8 = 1,
  atunci este bitul 6;
- în general, dacă
  (i*n+j)%8 = r,
  atunci este bitul 7-r.

FUNCŢIA BX44

```
unsigned belem(char [], int, int, int);

int relechiv(char ta[], int n)
/* returneaza 1 daca relatia pastrata in tabloul ta pe biti este o relatie de
   echivalenta si zero in caz contrar */
{
 int i,j,k;

/* reflexivitate */
 for(i=0; i < n ; i++ )
     if(belem(ta,n,i,i) == 0 )
```

```
        return 0; /* relatia nu este reflexiva */

   for( i=0; i < n; i++ )
       for(j=0; j<n; j++ )
           if(belem(ta,n,i,j) != belem(ta,n,j,i))
               return 0; /* relatia nu este simetrica */

   for(i=0; i < n; i++)
       for( j=0; j < n; j++)
           if(belem(ta,n,i,j))
               for(k=0; k < n; k++ )
                   if(belem(ta,n,j,k))
                       if(belem(ta,n,i,k) == 0)
                           return 0;

  /* relatia este tranzitiva */
   return 1;
  }

  unsigned belem(char ta[], int n, int i,int j)
  /* - returneaza valoarea elementului a[i][j];
     - elementele matricei patratice a, de ordinul n, sînt pastrate pe biti in
       tabloul ta, folosind relatia de liniarizare. */
  {
   return ta[(i*n+j)/8]>> 7-(i*n+j)%8&1;
  }
```

Observaţie:

Funcţia *belem* returnează valoarea elementului a[i][j]. Aşa cum s-a indicat mai sus, elementul respectiv este păstrat într-un bit al elementului:

ta[(i*n+j)/8].

Selectarea acestui bit se face deplasînd spre dreapta valoarea acestui element cu:

7 - (i*n+j)%8

poziţii binare (prin aceasta bitul în cauză devine cel mai puţin semnificativ), iar apoi se face un *şi logic pe biţi* dintre rezultatul acestei deplasări şi 1 (în felul acesta se anulează toţi biţii exceptînd ultimul, care este chiar valoarea lui a[i][j]).

Această funcţie se poate realiza folosind cîmpuri de biţi. În acest scop, se foloseşte o reuniune cu două componente. Una este o structură care

defineşte fiecare bit al unui octet, iar cealaltă este o dată de tip *char*. Elementul ta[(i*n+j)/8] se atribuie datei de tip *char* şi apoi se selectează bitul dorit prin acces direct la el.

```
unsigned belem(char ta[], int n,int i, int j)
/* returnează elementul a[i][j] pastrat pe biti in tabloul ta */
{
 union {
     struct {
         unsigned b0:1;
         unsigned b1:1;
         unsigned b2:1;
         unsigned b3:1;
         unsigned b4:1;
         unsigned b5:1;
         unsigned b6:1;
         unsigned b7:1;
     } b;
     char x;
 } octet;

/* pastreaza octetul ce contine bitul de selectat in zona
 alocata reuniunii */
 octet.x = ta[(i*n+j)/8];

/* selecteaza bitul de ordin 7-(i*n+j)%8 */
 switch ( 7-(i*n+j)%8) {
     case 0:
         return octet.b.b0;
     case 1:
         return octet.b.b1;
     case 2:
         return octet.b.b2;
     case 3:
         return octet.b.b3;
     case 4:
         return octet.b.b4;
     case 5:
         return octet.b.b5;
     case 6:
         return octet.b.b6;
     case 7:
         return octet.b.b7;
 }
```

}

10.45 Să se scrie o funcţie care calculează compunerea a două relaţii.

Fie P şi Q două relaţii reprezentate prin matricele MP şi MQ.

Operatorul de compunere a două relaţii îl notăm prin caracterul punct.

Fie R compunerea celor două relaţii P şi Q:

R = P.Q

Operatorul de compunere a două relaţii se defineşte ca mai jos.

Fie P o submulţime a produsului cartezian:

AxB

şi Q o submulţime a produsului cartezian:

BxC

Atunci R este o submulţime a produsului cartezian:

AxC

care se defineşte astfel:

perechea

<a,c>

aparţine relaţiei R, dacă există un element b, în mulţimea B, aşa încît perechea:

<a,b>

aparţine relaţiei P, iar perechea:

<b,c>

aparţine relaţiei Q.

Deci, dacă expresiile:

aPb şi bQc

sînt adevărate, atunci este adevărată şi expresia:

aRc

Fie MR matricea care reprezintă relaţia R. Dacă MP este o matrice de ordinul m*n, iar MQ o matrice de ordinul n*s, atunci matricea MR este de ordinul m*s.

Matricea relaţiei MX (X= P, Q sau R) este păstrată pe biţi în tabloul

unidimensional tmx (x = p, q sau r).

FUNCŢIA BX45

```
unsigned belem(char [],int,int,int);

void sbelem(char [],int,int,int);

void prodrel(char tmp[],char tmq[],char tmr[],
                int m,int n,int s)
/* - calculeaza relatia:
        R = P.Q
   - relatia P se reprezinta prin matricea MP;
   - relatia Q se reprezinta prin matricea MQ;
   - relatia R se reprezinta prin matricea MR;
   - matricea MP este de ordinul m*n si se pastreaza pe biti in tabloul
     tmp;
   - matricea MQ este de ordinul n*s si se pastreaza pe biti in
     tabloul tmq;
   - matricea MR este de ordinul m*s si se pastreaza pe biti in
     tabloul tmr. */
{
 int i,j,k;

/* se anuleaza toate elementele matricei MR */
 for(i=0; i<=m*s/8; i++)
     tmr[i] = 0;

/* se seteaza elementele matricei MR */
 for(j=0; j<n; j++)
     for(i=0; i<m; i++)
         if(belem(tmp,n,i,j))

/* MP[i][j] = 1 */
             for(k=0; k< s; k++ )
                 if(belem(tmq,s,j,k))

/* MQ[j][k] = 1 */
/* elementul MR[i][k] se seteaza la valoarea 1 */
                     sbelem(tmr,s,i,k);
}
```

```
void sbelem(char ta[],int n, int i,int j)
/* - seteaza la 1 valoarea elementului a[i][j];
   - matricea a este pastrata in tabloul ta pe biti folosind relatia de
     liniarizare;
   - matricea a are n coloane. */
{
 ta[(i*n+j)/8] |= 1<< 7 -(i*n+j)%8;
}
```

Observaţie:

Expresia:

(1) 1<<7 - (i*n+j)%8

deplasează pe unu spre stînga cu un număr de poziţii binare egal cu valoarea expresiei:

(2) 7 - (i*n+j)%8

Se observă că dacă i*n+j este multiplu de 8, atunci expresia (2) are valoarea 7 şi deci unu se deplasează cu 7 poziţii binare spre stînga. Se obţine:

10000000

deci, bitul cel mai semnificativ al octetului rezultat, devine egal cu 1.

Dacă restul împărţirii lui i*n+j la 8 este 1, atunci expresia (2) are valoarea 6, iar expresia (1) generează valoarea:

1000000

Deci, în acest caz, bitul de ordinul 6 al rezultatului, devine egal cu 1.

În mod analog, dacă (i*n+j)%8 = 2, bitul de ordinul 5 al rezultatului devine egal cu 1, şi aşa mai departe.

Dacă (i*n+j)%8 = 7, atunci expresia (2) are valoarea zero, iar expresia (1) are valoarea 1. Deci, bitul de ordinul zero al rezultatului va deveni egal cu 1.

10.46 Să se scrie o funcţie care citeşte indicii elementelor a[i][j] diferite de zero ale matricei *a* care reprezintă o relaţie binară. Matricea este de ordinul m*n şi elementele se păstrează pe biţi, în tabloul *ta* unidimensional, folosind relaţia de liniarizare.

La început funcţia citeşte valorile lui *m şi n*. Apoi se citesc indicii *i şi j* pentru fiecare element nenul al matricei *a*.

FUNCŢIA BX46

```
void citrel(char ta[], int max,int *m, int *n)
/* - citeste pe m si n;
   - citeste indicii elementelor a[i][j] nenule ale matricei a ce reprezinta
     o relatie binara;
   - pastreaza matricea pe biti in tabloul ta folosind relatia de liniarizare.
*/
{
 int i,j;

 do { /* citeste pe m si n */
     *m=*n=0;
     if(pcit_int_lim("numar linii m= ",
                    1,max,m) == 0 ) {
         printf("s-a tastat EOF\n");
         exit(1);
     }
     if(pcit_int_lim("numar coloane n= ",1,max,
              n) == 0 ) {
         printf("s-a tastat EOF\n");
         exit(1);
     }
     if(*m * *n < max)
         break;
     printf("produsul m*n= %d\n",*m * *n );
     printf("depaseste valoarea maxima= %d\n",max);
 } while(1);
 for(i=0; i<= *m * *n/8; i++)
     ta[i] =0;

/* citeste indicii elementelor nenule si seteaza bitii corespunzatori in
   tabloul ta */
 for( ; ; ) {
     if(pcit_int_lim("linia i= ", 1,*m,&i) == 0 )
         return;
     if(pcit_int_lim("coloana j= ",1,*n,&j)== 0 )
         return;

/* seteaza elementul a[i-1][j-1] */
     ta[((i-1)* *n + j-1)/8] |= 1 <<
             7-((i-1) * *n+j-1) % 8;
 }
}
```

10.47 Să se scrie un program care calculează şi afişează compunerea a două relaţii definite pe mulţimea A, apoi stabileşte dacă relaţiile respective sînt de echivalenţă.

Relaţiile se introduc folosind funcţia *citrel* definită în exerciţiul precedent.

Apoi, se apelează funcţia *prodrel* care compune relaţiile respective. În final, se stabileşte dacă cele trei relaţii sînt de echivalenţă.

PROGRAMUL BX47

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

#include "bviii2.cpp" /* pcit_int     */
#include "bviii3.cpp" /* pcit_int_lim */
#include "bx44.cpp"   /* relechiv     */
#include "bx45.cpp"   /* prodrel      */
#include "bx46.cpp"   /* citrel       */

void afisrel(char *, char [],int );

#define MAX 128

main() /*- calculeaza si afiseaza compunerea a doua relatii;
         - stabileste daca relatiile sînt de echivalenta. */
{
 char a[MAX *(MAX/8)], b[MAX* (MAX/8)],
      ab[MAX *(MAX/8)];
 int m,n,p,q;

/* defineste relatia reprezentata prin tabloul a */
 citrel(a,MAX*MAX,&m,&n);

/* defineste relatia reprezentata prin tabloul b */
 citrel(b,MAX*MAX,&p,&q);
 if(!(m==n&& n==p && p==q )) {
    printf("se cere ca relatiile sa fie\
          reprezentate prin matrice patratice\n");
    exit(1);
 }
```

```
/* se calculeaza compunerea relatiilor a si b:
     ab = a.b */
 prodrel(a,b,ab,m,m,m);

/* afiseaza relatia reprezentata prin a */
 afisrel("relatia a ", a,m);

/* afiseaza relatia reprezentata prin b */
 afisrel("relatia b ", b , m );

/* afiseaza compunerea relatiilor a si b */
 afisrel("relatia ab= a.b ",ab, m);

/* stabileste care dintre relatii este de echivalenta */
 if(relechiv(a,m))
     printf("relatia a este de echivalenta\n");
 else
     printf("relatia a nu este de echivalenta\n");
 if(relechiv(b,m))
     printf("relatia b este de echivalenta\n");
 else
     printf("relatia b nu este de\
             echivalenta\n");
 if(relechiv(ab,m))
     printf("relatia ab este de\
             echivalenta\n");
 else
     printf("relatia ab nu este de\
             echivalenta\n");
}   /* sfirsit main */

void afisrel(char *p, char ta[],int n)
/* afiseaza relatia pastrata pe biti prin liniarizare in tabloul ta */
{
 int i,j,k,m;

/* afiseaza un antet */
 printf("\n\n\t\t%s\n\n",p);
 for(i=0; i< n; i++) {
     printf("linia: %d\n", i+1);
     k=1,m=0 ;
     for(j=0; j<n; j++,k++) {
         printf("%d ",
```

```
                ta[(i*n+j)/8]>> 7-(i*n+j)%8&1);
            if( k == 30 ) {
                printf("\n");
                k = 0;
                m++;
                if(m%20 == 0 ) {
                    printf("actionati o tasta pentru a\
                            continua\n");
                    getch();
                }
            }
        }
        printf("actionati o tasta pentru a\
                continua\n");
        getch();
    }
}
```

10.6. Tipul enumerare

Tipul *enumerare* permite programatorului să folosească nume sugestive pentru valori numerice. De exemplu, în locul numărului unei luni calendaristice este mai sugestiv să folosim denumirea lunii respective sau eventual o prescurtare:

ian - Pentru ianuarie în locul cifrei 1.

feb - Pentru februarie în locul cifrei 2.

şi aşa mai departe.

Un alt exemplu se referă la posibilitatea de a utiliza cuvintele FALS şi ADEVARAT pentru valorile 0 respectiv 1. În felul acesta se obţine o mai mare claritate în programele sursă, deoarece valorile numerice sînt înlocuite prin sensurile atribuie lor într-un anumit context.

În acest scop se utilizează tipul *enumerare*. Un astfel de tip se declară printr-un format asemănător cu cel utilizat în cadrul structurilor. Un prim format general este:

enum *nume* { *nume0,nume1,nume2,...,numek* } *d1,d2,...,dn;*

unde:

nume - Este numele tipului de enumerare introdus prin această declaraţie.

nume0,nume1,..., numek	- Sînt nume care se vor utiliza în continuare în locul valorilor numerice şi anume *numei* are valoarea *i*.
d1,d2,...,dn	- Sînt date care se declară de tipul *nume*.
	- Aceste date sînt similare cu datele de tip *int*.

Ca şi în cazul structurilor, în declaraţia de mai sus nu sînt obligatorii toate elementele. Astfel, poate lipsi nume, dar atunci va fi prezent cel puţin *d1*. De asemenea, poate lipsi în totalitate lista d1,d2,...,dn, dar atunci va fi prezent *nume*. În acest caz, se vor defini ulterior date de tip *nume* folosind un format de forma:

enum *nume d1,d2,...,dn;*

Exemple:

1. enum { ileg,ian,feb,mar,apr,mai,iun,iul,aug,sep,oct,nov,dec } luna;

Prin această declaraţie, numărul lunii poate fi înlocuit prin denumirea prescurtată a lunii respective. De exemplu, o atribuire de forma:

luna = 3

se poate înlocui cu una mai sugestivă:

luna = mar

deoarece, conform declaraţiei de mai sus, *mar* are valoarea 3.

În mod analog, o expresie de forma:

luna == 7

este identică cu expresia:

luna == iul

Dacă în locul declaraţiei de mai sus s-ar fi utilizat declaraţia de tip enumerare:

enum dl { ileg,ian,feb,mar,apr,mai,iun,iul,aug,sep,oct,nov,dec };

atunci putem declara ulterior data *luna* de tip *dl* astfel:

enum dl luna;

Data *luna* declarată în acest fel este o dată identică cu data *luna* declarată la început.

2. Fie tipul enumerare *Boolean* declarat astfel:

enum Boolean { false, true};

Declarăm data *bisect* de tip Boolean:

enum Boolean bisect;

Atribuirea:

bisect = an%4 == 0 && an%100 || an%400 == 0;

atribuie variabilei *bisect* valoarea 1 sau 0, după cum anul definit de variabila *an* este bisect sau nu (se presupune că anul aparţine intervalului [1600,4900]).

În continuare se pot folosi expresii de forma:

bisect == false

sau

bisect == true

3. Fie *f* o funcţie care returnează numai două valori: 0 sau 1. O astfel de funcţie se consideră, de obicei, că returnează o valoare booleană. Atunci, antetul ei poate fi definit astfel:

   enum Boolean f(...)

   În continuare se pot folosi expresii de forma:

   f(...) == false

sau

   f(...) == true

La declararea tipurilor enumerare se poate folosi cuvîntul cheie *typedef*, ca şi în cazul tipurilor utilizator definite prin *struct*. În acest caz vom utiliza o declaraţie de forma:

typedef enum *nume* { *nume0,nume1,...,numek* } *nume_tip*;

În continuare, se pot declara mai simplu date de tipul enumerare *nume* folosind:

nume_tip d1,d2,...,dn;

În acest caz *nume* este opţional şi de obicei lipseşte. Folosind această convenţie, exemplele de mai sus pot fi scrise astfel:

```
typedef enum { ileg, ian,feb,mar,apr,mai,iun,
        iul,aug,sep,oct,nov,dec } DL;
DL luna;
typedef enum { false, true} Boolean;
Boolean bisect;
```

sau antetul funcţiei *f*:

Boolean f(...)

Tipul enumerare poate fi declarat impunînd valori altele decît cele care rezultă implicit. În acest caz, *numei* se va înlocui cu:

numei = eci

unde:

eci - Este o expresie constantă.

Numele *numei* va avea ca valoare, valoarea expresiei *eci*. Dacă numelui următor nu i se atribuie o valoare, atunci acesta va avea ca valoare, valoarea lui *numei* mărit cu 1.

Folosind această observaţie, modificăm tipul DL astfel:

```
typedef enum {ian=1,feb,mar,apr,mai,iun,iul,aug,sep,oct,nov,dec } DL;
```

Menţionăm că datele de acest tip nu sînt supuse la controale din partea compilatorului C şi de aceea se pot scrie expresii care să nu corespundă scopului pentru care s-au definit datele respective.

De exemplu, fie:

```
DL d1,d2,d3;
```

Expresiile de mai jos nu sînt interpretate ca eronate de compilator:

```
d3 = d1 + d2;
d3 = d1 * d2;
d3 = d1/d2; etc.
```

Aceasta, deoarece datele de tip enumerare sînt tratate ca simple date de tip *int*.

10.7. Date definite recursiv

Fie declaraţia:

```
struct nume {
    declaratii
};
```

Declaraţiile incluse între acolade definesc componentele datelor structurate de tip *nume*, tip definit prin declaraţia struct. Aceste declaraţii pot defini date de diferite tipuri predefinite sau utilizator, dar diferite de tipul *nume*. În schimb, se pot defini pointeri spre date de tipul *nume*. Deci, o declaraţie de forma:

struct *nume* {
    declaratii
    struct *nume* **nume1*;
    declaratii
};

este corectă, în timp ce declaraţia:

struct *nume* {
    declaratii
    struct *nume nume1*;
    declaratii
};

este eronată.

Un tip utilizator, se spune că este *direct recursiv* dacă el are cel puţin o componentă care este de tip *pointer spre el însuşi*.

Exemplu:

```
struct tnod {
 char cuvint[100];
 int nr;
 struct tnod *urm;
} ;
```

Tipul utilizator *tnod* conţine 3 componente:

- tabloul *cuvint* de tip *char* de 100 elemente;
- variabila *nr* de tip *int*;
- pointerul *urm* spre tipul *tnod*.

tnod este un tip direct recursiv deoarece conţine un pointer spre el însuşi.

În continuare, putem defini obişnuit date de tipul *tnod*.

Evident, tipurile direct recursive pot fi denumite folosind construcţia *typedef*. Tipul de mai sus îl vom numi TNOD folosind declaraţia:

```
typedef struct tnod {
 char cuvint[100];
 int nr;
 struct tnod *urm;
} TNOD;
```

În continuare putem defini date folosind tipul TNOD:

TNOD nod, *p;

O altă posibilitate în definirea tipurilor utilizator este acela în care un tip *t1* conţine un pointer spre tipul *t2*, iar acesta, la rîndul lui, conţine un pointer spre tipul *t1*. În acest caz se spune că tipul *t1* este *indirect recursiv*.

Să observăm că la declararea tipului *t1*, se utilizează o declaraţie de forma:

struct *t2* ***nt2**;

Aceasta este posibil dacă tipul *t2* este definit înainte de tipul *t1*. Să presupunem că acest lucru are loc. Dar la declararea tipului *t2* constatăm că acesta conţine o declaraţie de forma:

struct *t1* ***nt1**;

Aceasta la rîndul ei poate fi scrisă dacă înaintea declaraţiei tipului *t2* este prezentă declaraţia care defineşte tipul *t1*. Se ajunge în felul acesta la o contradicţie. Pentru a o elimina, s-a introdus în limbajul C declaraţia de tip incompletă.

Aceasta are formatul:

struct *nume*;

Această declaraţie poate fi plasată într-un fişier sursă înainte de a defini tipul *nume* sau în orice fişier sursă în care tipul *nume* nici nu este definit. După declaraţia de tip incompletă, se pot declara pointeri spre tipul respectiv. În felul acesta, tipurile *t1 şi t2* de mai sus, se vor declara astfel:

```
struct  t1; /* - declaratie incompleta pentru tipul t1;
              - in continuare se poate declara tipul t2 */
struct t2 {
    declaratii
    struct t1 *nt1;
    declaratii
};
struct t1 { /* declaratia completa a lui t1 */
    declaratii
    struct t2 *nt2;
    declaratii
};
```

Un tip se spune că este *recursiv* dacă el este direct sau indirect recursiv.

O *dată este recursivă* dacă ea este o dată de un tip recursiv.

În general, un tip *t* este *indirect recursiv* dacă există un şir de tipuri *ti*:

t1,t2,t3,...,tn

aşa încît tipul *ti* conţine cel puţin o componentă care este pointer spre tipul *tj(j=i+1)*, pentru i = 1,2,...,n-1, iar *t1 şi tn* sînt ambele de tipul *t*.

Datele recursive se utilizează într-o serie de aplicaţii bazate pe definirea şi prelucrarea listelor, arborilor, tabelelor de dispersie etc.

Ele se folosesc cu succes în prelucrarea dinamică a datelor.

În multe aplicaţii se utilizează date structurate de un anumit tip, care se repetă de un număr variabil de ori, număr care se precizează la fiecare execuţie a programului.

În astfel de situaţii se poate defini un tablou de structuri a cărui număr de elemente să fie egal cu o valoare maximă precizată de utilizator. Acest mod de lucru nu este cel mai indicat atunci cînd numărul datelor diferă mult de la o execuţie la alta. De asemenea, pot să se ivească situaţii în care acest maxim este greu de evaluat, ca să nu mai vorbim de faptul că el poate fi evaluat greşit. De aceea, în astfel de cazuri este mult mai util să se păstreze datele respective în memoria *heap*, rezervîndu-se memorie pentru ele la execuţie, pe măsură ce este nevoie. Prelucrarea datelor respective implică posibilitatea de a trece simplu de la o dată la alta. În cazul tablourilor,aceasta se asigură cu ajutorul indicilor. În noua situaţie se poate trece de la o dată păstrată în memoria *heap* la o altă dată de acelaşi tip, dacă se utilizează un pointer spre tipul comun datelor respective şi care să fie componentă a datei respective. Deci, se ajunge la un tip de forma:

```
typedef struct nume {
        declaratii
        struct nume *urm;
        declaratii
} nume_tip;
```

În felul acesta se ajunge la necesitatea de a utiliza date recursive.

Gestiunea dinamică a lor mai are încă un avantaj şi anume: astfel de date pot fi eliminate uşor din memoria *heap* cînd nu este nevoie de ele şi zona eliberată în acest fel poate fi realocată altor date de acelaşi tip sau în alte scopuri.

Exerciţii:

10.48 Să se scrie o funcţie care citeşte un cuvînt şi-l păstrează în memoria *heap*. Prin *cuvînt* înţelegem o succesiune de litere mici sau mari.

Funcţia returnează adresa de început a zonei din memoria *heap* în care se păstrează cuvîntul citit sau zero la întîlnirea sfîrşitului de fişier.

FUNCŢIA BX48

```
char *citcuv()
/* - citeste un cuvint si-l pastreaza in memoria heap;
   - returneaza pointerul spre cuvintul respectiv sau zero la sfirsit
     de fisier. */
{
 int c,i;
 char t[255];
 char *p;

/* salt peste caractere care nu sînt litere */
 while((c=getchar()) <'A' || (c > 'Z' &&
       c < 'a') || c > 'z')
     if( c == EOF )
         return 0; /* s-a tastat EOF */

/* se citeste cuvintul si se pastreaza in t */
 i = 0;
 do {
       t[i++] = c;
 } while((c=getchar()) >= 'A' &&
          c<='Z'||c>= 'a' && c <= 'z');
 if( c == EOF )
     return 0;
 t[i++] = '\0';

/* se pastreaza cuvintul in memoria heap */
 if((p = (char *)malloc(i)) == 0 ) {
     printf("memorie insuficienta\n");
     exit(1);
 }
 strcpy(p,t);
 return p;
}
```

10.49 Să se scrie o funcţie care citeşte cuvintele dintr-un text şi determină numărul de apariţii al fiecărui cuvînt din textul respectiv.

Această problemă poate fi rezolvată în mai multe moduri. Prezentăm mai jos o metodă simplă, care însă nu este şi eficientă. Metode mai eficiente vor fi prezentate în alte capitole.

Înainte de toate să observăm că problema se poate rezolva folosind un tablou de pointeri spre *char*, dacă putem indica o valoare maximă pentru

numărul de cuvinte diferite din text. În acest caz se defineşte tabloul:

char *tp[maxcuv];

şi un indice *itp* care are ca valoare indicele primului element liber din tablou.

De asemenea, vom utiliza un tablou pentru a număra apariţiile fiecărui cuvînt:

int nc[maxcuv];

Procesul de calcul se realizează astfel:

1. itp = 0.
2. Se apelează funcţia *citcuv* pentru a citi cuvîntul curent:

   p = citcuv();

   Dacă p = 0, s-a întîlnit sfîrşitul de fişier şi procesul de calcul se întrerupe.

   Altfel se trece la pasul următor.
3. Se stabileşte dacă cuvîntul citit (spre care pointează p) a fost deja întîlnit în text.

   În acest scop se compară cuvintele spre care pointează elementele:

   tp[0],tp[1],...,tp[itp-1]

   cu cel spre care pointează p.

   În caz afirmativ, se trece la pasul 4, altfel la pasul 5.
4. Dacă cuvîntul spre care pointează *p* coincide cu cel spre care pointează tp[j], atunci se incrementează nc[j] (numărul de apariţii al cuvîntului respectiv); apoi se elimină cuvîntul, citit la punctul 2, din memoria *heap* şi procesul de calcul se reia de la pasul 2.
5. Deoarece cuvîntul spre care pointează *p* nu a fost încă întîlnit în textul de intrare, pointerul *p* se păstrează şi numărul de apariţii al cuvîntului respectiv se face egal cu 1:

   tp[itp] = p;

   nc[itp] = 1;

   apoi se măreşte *itp* cu o unitate:

   itp = itp + 1;

   procesul de calcul se reia de la pasul 2.

Funcţia utilizează un tablou de structuri de tipul T declarat astfel:

```
typedef struct {
 char *tp;
 int nc;
} T;
```

Ea are un parametru care este un tablou de tipul T. Un alt parametru al ei este *maxcuv*.

Funcţia returnează numărul cuvintelor distincte citite (*itp*).

FUNCŢIA BX49

```
int frecvcuv(T t[], int maxcuv)
/* - citeste un text si determina frecventa de aparitie a fiecarui cuvint
     citit;
   - se admit cel mult maxcuv cuvinte diferite;
   - functia returneaza numarul cuvintelor distincte intilnite in text */
{
 char *p;
 int itp;
 int i;

 for(itp =0; itp < maxcuv; ) {

/* citeste cuvintul curent */
     if(( p = citcuv() ) == 0 )

/* s-a intilnit sfirsitul de fisier */
          return itp;

/* se stabileste daca cuvintul citit a fost deja intilnit */
     for(i = 0; i < itp; i++)
        if(strcmp(t[i].tp,p) == 0 )
           break;
     if( i < itp ) {

  /* cuvint deja intilnit */
        t[i].nc++;
        free(p);
     }
     else { /* cuvint nou */
        t[itp].tp = p;
        t[itp++].nc = 1;
```

```
    }
  }
  return itp;
}
```

10.50 **Să se scrie un program care afişează frecvenţa de apariţie a cuvintelor unui text.**

PROGRAMUL BX50

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <string.h>
#include <alloc.h>

typedef struct {
 char *tp;
 int nc;
} T ;

#include "bx48.cpp"
#include "bx49.cpp"

#define MAXCUV 100

main() /* afiseaza frecventa de aparitie a cuvintelor unui text */
{
 T tab[MAXCUV];
 int ncuv,i;

 ncuv = frecvcuv(tab,MAXCUV);
 for(i=0; i < ncuv; i++) {
    printf("%s\t%d\n",tab[i].tp,tab[i].nc);
    if((i+1)%23 == 0 ) {
       printf("actionati o tasta pentru a\
             continua\n");
       getch();
    }
 }
}
```

10.51 Să se modifice funcţia *frecvcuv* definită în exerciţiul 10.49, în aşa fel

încît în loc de tabloul de tip T să se utilizeze date de tip recursiv.

În acest caz vom folosi tipul recursiv:

```
typedef struct tr {
 char *tp;
 int nc;
 struct tr *urm;
} TR;
```

Pentru fiecare cuvînt se va păstra, în memoria heap, o dată de tip TR. Pointerului *urm* i se atribuie adresa de început a zonei din memoria heap în care se păstrează cuvîntul următor citit din textul de intrare. El are valoarea zero pentru data corespunzătoare ultimului cuvînt citit.

Pentru a gestiona simplu datele păstrate în memoria heap, va fi util să existe două variabile de tip pointer spre TR, una să pointeze spre prima dată păstrată în memoria heap, iar cealaltă spre ultima. Vom numi *prim şi ultim* aceste variabile.

Funcţia realizează paşii de mai jos:

1. prim = ultim = 0.
2. Se apelează funcţia *citcuv* pentru a citi cuvîntul curent:

   p = citcuv();

   Dacă p = 0, atunci se revine din funcţie cu valoarea variabilei *prim*; altfel se trece la pasul următor.
3. Se stabileşte dacă cuvîntul citit (spre care pointează p) a fost deja întîlnit în text.

   În acest scop se compară cuvintele spre care pointează pointerul *tp* din datele de tip TR păstrate în memoria heap, cu cel spre care pointează *p*.

   În caz afirmativ se incrementează numărătorul cuvîntului respectiv, se eliberează zona ocupată de cuvîntul curent citit şi procesul se reia de la pasul 2.

   Altfel, se rezervă zonă pentru o dată de tip TR şi fie *q* adresa ei de început. Pointerului *urm*, corespunzător ultimei date de tip TR păstrate în memoria heap, i se atribuie valoarea lui *q*.

   Apoi nc = 1 şi urm = 0 pentru data spre care pointează *q*.

   Cum aceasta devine ultima dată păstrată în memoria heap, se face ultim=q şi apoi se revine la pasul 2.

FUNCŢIA BX51

```
TR *pfrecvcuv()
/* citeste un text si pastreaza in memoria heap date de tip TR pentru
   cuvintele distincte din text, determina frecventa acestora si retur-
   neaza pointerul spre data corespunzatoare primului cuvint din text */
{
 TR *prim, *ultim,*q;
 char *p;

 prim = ultim =0;
 for( ; ; ) {

/* citeste cuvintul curent */
    if((p = citcuv() ) == 0)

/* s-a intilnit sfirsitul de fisier */
        return prim;

/* stabileste daca cuvintul curent citit a fost deja intilnit in text */
    q = prim; /* cautarea se face incepind cu primul cuvint */
    while ( q ) { /* daca q != 0 inseamna ca pointeaza spre o
                      data din memoria heap */
        if(strcmp( q->tp,p) == 0)
            break; /* cuvintul a mai fost intilnit in text */

/* - se trece la data corespunzatoare cuvintului urmator citit din textul de
     intrare;
   - adresa acestei date este valoarea pointerului urm al datei spre care
     pointeaza q;
   - aceasta se poate exprima cu ajutorul expresiei:
        q -> urm;
   - aceasta adresa se atribuie lui q si in felul acesta se poate relua ciclul */

        q = q -> urm;
    }
    if(q) { /* daca q != 0, cuvîntul a mai fost intilnit in textul
               de intrare */
        q -> nc++; /* se incrementeaza numarul de aparitii al
                      cuvintului */
        free(p); /* se elibereaza zona ocupata de cuvint */
    }
    else { /* cuvintul este nou */
```

```
/* se rezerva zona pentru o data de tip TR */
          q =(TR *)malloc(sizeof(TR));
          if(q == 0 ) { /* nu se poate aloca memoria heap */
              printf("memorie insuficienta\n");
              exit(1);
          }

/* se pastreaza pointerul spre cuvintul nou */
          q -> tp = p;

/* nc = 1 */
          q -> nc = 1;
          if(ultim ) /* daca ultim != 0, cuvintul curent nu este
                            primul cuvint citit */
/* ultim pointeaza spre data din memoria heap corespunzatoare
   cuvintului precedent pastrat, deci
       ultim -> urm = q
*/
              ultim -> urm = q;

/*- q -> urm = 0, deoarece data spre care pointeaza q corespunde
   ultimului cuvint citit si pastrat in memoria heap;
  - din aceleasi motive
         ultim = q
*/
          q -> urm = 0;
          ultim = q;
          if(prim == 0 ) /* cuvintul curent este primul cuvint
                                citit */
              prim = q;
      } /* sfirsit else */
  } /* sfirsit for */
}
```

10.52 Să se scrie un program care citeşte şi afişează frecvenţa de apariţie a cuvintelor din textul citit, folosind funcţia *pfrecvcuv* definită în exerciţiul precedent.

În programul de faţă nu mai este necesar să ne limităm la 100 de cuvinte, cum am făcut în programul din exerciţiul 10.50. Datele se păstrează în memoria heap pe măsură ce este nevoie de ele.

PROGRAMUL BX52

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <string.h>
#include <alloc.h>

typedef struct tr {
 char *tp;
 int nc;
 struct tr *urm;
} TR;

#include "bx48.cpp" /* citcuv    */
#include "bx51.cpp" /* pfrecvcuv */

main() /* afiseaza frecventa de aparitie a cuvintelor unui text */
{
 TR *p;
 int i;

 p = pfrecvcuv();
 i = 1;

 while( p ) {
     printf("%s\t%d\n",p -> tp, p -> nc);
     if(i%23 == 0 ) {
         printf("actionati o tasta pentru a\
                   continua\n");
         getch();
     }

     i++;
     p = p -> urm;
 }
}
```

Observaţie:

Datele de tip TR, păstrate în memoria *heap*, sînt structuri recursive care formează o mulţime ordonată. Există o *primă* structură şi fiecare structură are un element următor, exceptînd ultima. De aceea, între elementele acestei mulţimi de structuri este stabilită o relaţie de ordine. Relaţia

respectivă se defineşte printr-un pointer care este componentă a tipului comun al acestor structuri.

O astfel de mulţime se spune că formează o *listă simplu înlănţuită.*

11. LISTE

Lista este o mulţime *dinamică*, înţelegînd prin aceasta că ea are un număr variabil de elemente. La început lista este o mulţime vidă. În procesul execuţiei programului se pot adăuga elemente noi listei şi totodată pot fi eliminate diferite elemente din listă de care nu mai este nevoie.

Elementele unei liste sînt date de un acelaşi tip. În forma cea mai generală, tipul comun elementelor unei liste este un tip utilizator. Deci,elementele unei liste sînt structuri de un acelaşi tip. Acest fapt ne conduce la ideea de a organiza o listă sub formă de tablou. Acest lucru este posibil, dar de obicei nu este eficient, deoarece lista are o natură dinamică, ori tablourile în limbajul C nu sînt dinamice. La compilare este necesar să se definească un maxim pentru numărul elementelor unui astfel de tablou, maxim care uneori nu se poate stabili dinainte. Se poate adopta o soluţie de compromis dacă se păstrează elementele listei în memoria *heap*, iar într-o memorie statică sau pe stivă se alocă un tablou de pointeri spre elementele listei. În acest caz, elementele tabloului de pointeri ocupă fiecare 16 biţi sau 32 de biţi dacă pointerii respectivi sînt de tip *far*. Tabloul de pointeri se va dimensiona la un număr de elemente suficient de mare. Avantajul păstrării elementelor listei în memoria *heap* rezultă din faptul că, supradimensionarea tabloului de pointeri nu conduce la un consum prea mare de memorie deoarece elementele tabloului sînt pointeri şi nu însăşi elementele listei. În acelaşi timp, memoria *heap* este gestionată optim, deoarece ea conţine, în fiecare moment al execuţiei programului, numai elementele necesare din listă. Cu toate acestea, există situaţii în care este dificil a evalua numărul maxim al elementelor unei liste, precum şi cazuri cînd numărul lor diferă mult de la execuţie la execuţie. De aceea, apare problema de a organiza altfel mulţimile de tip listă. Se obişnuieşte să se ordoneze elementele unei liste folosind pointeri care intră în compunerea elementelor listei. Datorită acestor pointeri, elementele listei devin structuri recursive. Listele organizate în acest fel se numesc *liste înlănţuite*.

În concluzie, o mulţime dinamică de structuri recursive de acelaşi tip, pentru care sînt definite una sau mai multe relaţii de ordine cu ajutorul unor pointeri din compunerea structurilor respective, se numeşte *listă înlănţuită*.

Elementele unei liste, de obicei, se numesc *noduri*.

Dacă între nodurile unei liste există o singură relaţie de ordine, atunci lista se numeşte *simplu înlănţuită*. În mod analog, lista este *dublu înlănţuită* dacă între nodurile ei sînt definite două relaţii de ordine.

În general, vom spune că o listă este *n-înlănţuită* dacă între nodurile ei

sînt definite *n* relaţii de ordine.

În legătură cu listele înlănţuite se au în vedere unele operaţii de interes general:

a. crearea unei liste înlănţuite;

b. accesul la un nod oarecare al unei liste înlănţuite;

c. inserarea unui nod într-o listă înlănţuită;

d. ştergerea unui nod dintr-o listă înlănţuită;

e. ştergerea unei liste înlănţuite.

11.1. Lista simplu înlănţuită

Aşa cum s-a arătat mai sus, între nodurile unei liste simplu înlănţuite este definită o singură relaţie de ordonare. De obicei această relaţie este cea de succesor, adică fiecare nod conţine un pointer a cărui valoare reprezintă adresa nodului *următor* din listă. În mod analog se poate defini relaţia de *precedent*. În cele ce urmează ne vom mărgini numai la liste simplu înlănţuite pentru care nodurile satisfac relaţia de succesor.

Într-o astfel de listă există totdeauna un nod şi numai unul care nu mai are următor (succesor), precum şi un nod care nu este următorul (succesorul) nici unui alt nod. Aceste noduri formează *capetele* listei simplu înlănţuite.

Pentru a gestiona nodurile unei liste simplu înlănţuite, vom utiliza doi pointeri spre cele două capete. Numim *prim* pointerul spre nodul care nu este următorul nici unui nod al listei şi cu *ultim* pointerul spre nodul care nu are următor (succesor) în listă. Aceşti pointeri vor fi utilizaţi în toate funcţiile care le vom avea în vedere în prelucrarea listelor simplu înlănţuite. Ei pot fi definiţi fie ca variabile globale, fie ca parametri pentru funcţiile de prelucrare a listei. Se alege alternativa cu variabile globale în cazul în care nu se gestionează mai multe liste simplu înlănţuite în acelaşi program. Dimpotrivă, pointerii respectivi se vor transfera prin parametri în cazul în care programul gestionează mai multe liste simplu înlănţuite.

În cele ce urmează vom considera cazul în care pointerii *prim* şi *ultim* sînt variabile globale.

Tipul unui nod într-o listă simplu înlănţuită se poate defini folosind o declaraţie de forma:

typedef struct *tnod* {
    declaraţii

```
        struct tnod *urm;
    } TNOD;
```

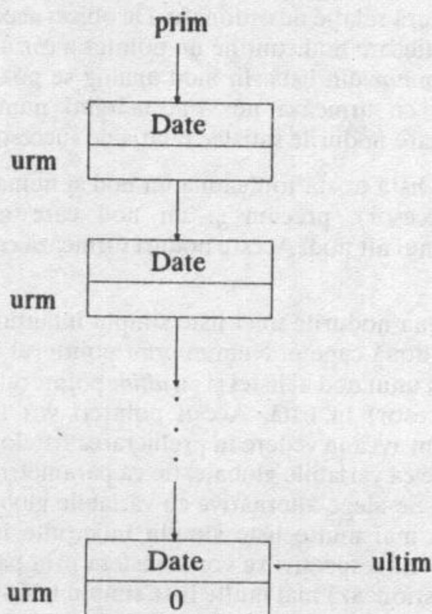
Pointerii *prim* şi *ultim* se declară în afara oricărei funcţii prin:

```
TNOD *prim,*ultim;
```

De obicei, ei vor fi declaraţi înaintea definirii funcţiei *main* a programului.

Pointerul *urm* defineşte relaţia de succesor pentru nodurile listei. Pentru fiecare nod, el are ca valoarea adresa nodului următor din listă. Excepţie de la această regulă o constituie nodul spre care pointează variabila *ultim*. În acest caz *urm* are valoarea zero (pointerul nul).

În paragrafele următoare vom construi funcţii care să realizeze operaţiile a-e, amintite mai sus, pentru listele simplu înlănţuite.



Listă simplu înlănţuită

11.1.1. Crearea unei liste simplu înlănţuite

La crearea unei liste simplu înlănţuite se realizează următoarele:

1. Se inițializează pointerii *prim* și *ultim* cu valoarea zero, deoarece lista la început este vidă.
2. Se rezervă zonă de memorie în memoria *heap* pentru nodul curent.
3. Se încarcă nodul curent cu datele curente, dacă există și apoi se trece la pasul 4.

   Altfel lista este creată și se revine din funcție.
4. Se atribuie pointerului

   ultim -> urm

   adresa din memoria *heap* a nodului curent, dacă lista nu este vidă.

   Altfel se atribuie lui *prim* această adresă.
5. Se atribuie lui *ultim* adresa nodului curent.
6. ultim -> urm = 0
7. Procesul se reia de la punctul 2 de mai sus pentru a adăuga un nod nou la listă.

Pentru a încărca datele curente în nod (punctul 3) vom apela o funcție care este specifică aplicației. Această funcție poate să aibă un nume dinainte precizat sau să fie transferată printr- un parametru la funcția de creare a listei. Acest parametru va fi un pointer spre funcția respectivă. Soluția cu parametru se alege de obicei cînd programul prelucrează mai multe liste. În cazul de față am presupus că programul prelucrează o singură listă și de aceea în loc să utilizăm un parametru pointer, vom considera că datele se încarcă prin funcția de nume *incnod*. Ea returnează una din următoarele trei valori: 0, 1 și -1 și anume:

0	- La eroare.
1	- La încărcare normală a datelor în nodul curent.
-1	- Cînd nu mai sînt date de încărcat în nod.

De exemplu, dacă funcția *incnod* citește datele pe care le încarcă în nod dintr-un fișier, atunci ea va returna valoarea -1 la întîlnirea sfîrșitului de fișier (nu mai sînt date de încărcat în nod).

Funcția *incnod* are un singur parametru și anume adresa din memoria *heap* rezervată pentru nodul curent, deci acest parametru este un pointer spre tipul comun nodurilor structurii, adică spre TNOD.

Din cele de mai sus rezultă că funcția *incnod* are prototipul:

int *incnod(TNOD *p)*;

O altă funcţie specifică aplicaţiei concrete şi care se apelează la crearea listei este cea care eliberează zona de memorie rezervată nodului curent. Vom numi *elibnod* această funcţie. Ea are prototipul:

void *elibnod(TNOD *p)*;

Această funcţie se apelează după un apel al funcţiei *incnod* în care aceasta nu a încărcat date în nodul curent (s-a revenit din ea cu valoarea 0 sau -1). În acest caz există zonă de memorie rezervată în memoria *heap* pentru nodul curent, dar la apelul funcţiei *incnod* nu s-au încărcat date ori s-au încărcat date parţial. De aceea, în acest caz vom elibera zona de memorie rezervată apelînd *elibnod*.

Funcţia pentru crearea listei simplu înlănţuite o numim *crelist*. Ea returnează una din valorile 0 sau -1 şi anume:

0 - La eroare.

-1 - La crearea normală a listei.

Ţinînd seama de cele de mai sus, definim funcţia *crelist* astfel:

```
int crelist() /*- creaza o lista simplu inlantuita;
                 - returneaza:
                        0 - la eroare;
                       -1 - creare normala. */
{
 extern TNOD *prim,*ultim;
 int i,n;
 TNOD *p;

 n=sizeof(TNOD);
 prim=ultim=0; /* initial lista este vida */

/* se rezerva n octeti pentru nod in memoria heap */
 for(;;){
     if((p=(TNOD *)malloc(n))==0){
         printf("memorie insuficienta la crearea\
                listei\n");
         exit(1);
     }

/* se incarca date in nod */
     if((i=incnod(p))!=1){
         elibnod(p); /* se elibereaza zona rezervata pentru nod
                        deoarece nu s-au incarcat date in nod */
         return i;
```

```
    }
 /* se inlantuie nodul in lista */
    if(ultim!=0)
 /* lista nu este vida */
        ultim -> urm=p;
    else
 /* lista vida */
        prim=p;

    ultim=p;
    ultim -> urm=0;
  } /* sfirsit for */
 } /* sfirsit crelist */
```

Aşa cum s-a arătat înainte, o listă poate fi organizată păstrînd elementele ei în memoria *heap*, iar adresele lor într-o tabelă de pointeri.

Dăm mai jos o variantă a funcţiei *crelist* care să creeze o listă după acest principiu. În acest caz, tabloul de pointeri va fi global şi are un număr maxim de elemente.

Numim *tpnod* acest tablou. De asemenea, vom nota cu *itpnod* variabila globală care are ca valoare indicele primului element al tabloului *tpnod* care este liber. Cu alte cuvinte, tpnod[itpnod -1] este pointerul spre ultimul nod adăugat la listă. Iniţial *itpnod* are valoarea zero.

Tipul nodurilor nu mai este necesar să fie recursiv, deoarece în acest caz ordonarea se realizează prin indici:

- primul nod are indicele zero;
- al doilea nod are indicele 1;

  ...

- ultimul nod are indicele itpnod-1.

De aceea, în locul tipului TNOD vom folosi tipul:

typedef struct {
    declaratii
} *TTNOD*;

Tabloul *tpnod* se declară astfel:

TTNOD *tpnod[MAX];

unde:

MAX - Este o constantă simbolică în prealabil definită printr- o construcţie *#define* şi valoarea ei se va alege aşa încît să nu fie depăşită de numărul elementelor listei.

Variabila *itpnod* este de tip *int*:

int itpnod;

Indicăm mai jos funcţia pentru crearea listei cu utilizarea tabloului *tpnod*. Ea utilizează funcţiile *incnod* şi *elibnod* ca în cazul funcţiei *crelist*, fiind analogă cu aceasta.

```
int tpcrelist() /*- creaza o lista folosind un tablou de pointeri
                    care contine adresele nodurilor listei;
                  - functia returneaza:
                        0 - la eroare;
                       -1 - la creare normala. */
{
 extern TTNOD *tpnod[];
 extern int itpnod;
 int i,n;
 TTNOD *p;

 n=sizeof(TTNOD);
 for(itpnod=0;itpnod<MAX;itpnod++){
     if((p=(TTNOD *)malloc(n))==0){
         printf("memorie insuficienta\n");
         exit(1);
     }

/* se incarca date in nod */
     if((i=incnod(p))!=1){
         elibnod(p);
         return i;
     }
     tpnod[itpnod]=p; /* pastreaza adresa nodului in tabloul de
                         pointeri */
 } /* sfirsit for */
}
```

11.1.2. Accesul la un nod al unei liste simplu înlănţuite

Putem avea acces la nodurile unei liste simplu înlănţuite începînd cu nodul spre care pointează variabila globală *prim* şi trecînd apoi pe rînd de la un nod la altul, folosind pointerul *urm*.

Există cazuri în care dorim acces la un nod anumit al listei. În acest caz, este necesar să definim modul de indentificare al nodului la care dorim să avem acces. Un mod simplu este acela de a indica numărul de ordine al nodului la care se doreşte acces, de exemplu, se doreşte acces la al *n*-lea nod al listei. Cum lista este o mulţime dinamică, acest mod de a defini accesul la un nod al ei nu este totdeauna cel mai nimerit. O altă metodă este aceea, de a avea o dată componentă a nodurilor, care să aibă valori diferite, pentru noduri diferite. În acest caz se poate defini accesul la nodul din listă pentru care data respectivă are o valoare dată. O astfel de dată, care este componentă a nodurilor unei liste şi are valori distincte pentru noduri diferite ale unei liste se numeşte *cheie*.

Cheia poate fi o dată de un tip oarecare.

În cazul de faţă vom considera cheia de tip *int*. Cititorul poate modifica simplu funcţia de mai jos pentru a utiliza chei de alte tipuri: *long*, *char*, *double* etc.

Funcţia returnează pointerul spre nodul căutat sau zero în cazul în care lista nu conţine un nod a cărui cheie să aibă valoarea indicată de parametrul ei.

Nodurile listei au tipul definit astfel:

```
typedef struct tnod{
        declaratii
        int cheie;
        declaratii
        struct tnod *urm;
}TNOD;
```

Putem acum defini funcţia de căutare ca mai jos:

```
TNOD *cnci(int c) /* - cauta un nod al listei pentru care cheie=c;
                     - returneaza pointerul spre nodul determi-
                       nat in acest fel sau zero daca nu exista nici
                       un nod asa incit cheia sa aiba valoarea c. */
{
 extern TNOD *prim;
 TNOD *p;

 p=prim; /* cautarea incepe cu nodul spre care pointeaza variabila
            prim */
 while(p!=0){
     if(p -> cheie==c)
         return p; /* s-a gasit un nod pentru care cheie=c */
```

```
        p=p ->urm;  /* se trece la nodul urmator din lista */
    }
   /* in acest punct se ajunge cind nu exista un nod in lista cu proprietatea
     ca cheie=c */
   return 0;
   }
```

O funcţie similară se poate construi şi pentru cazul în care la implementarea listei se utilizează tabloul de pointeri *tpnod* (vezi paragraful precedent).

În acest caz tipul nodurilor listei se declară astfel:

typedef struct {
    declaratii
    int *cheie;*
    declaratii
} *TTNOD;*

Funcţia caută nodul din listă parcurgînd-o cu ajutorul indicelui care variază de la zero pînă la *itpnod-1* inclusiv.

```
TTNOD *tpcnci(int c)
/*- cauta un nod al listei pentru care cheie=c;
  - returneaza pointerul spre nodul respectiv sau zero daca nu exista in
    lista un astfel de nod */
{
 extern TTNOD *tpnod[];
 extern int itpnod;
 int i;

 for(i=0;i<itpnod;i++)
     if(tpnod[i] -> cheie==c)
         return tpnod[i];
 return 0;
}
```

Ambele funcţii parcurg elementele listei, nod cu nod, fie pînă la întîlnirea nodului căutat, fie pînă la sfîrşitul listei dacă nu există în listă un nod a cărui cheie să aibă valoarea cerută. Această metodă de căutare se numeşte *căutare liniară*. Ea este o metodă foarte simplă de căutare dar nu este eficientă şi de aceea se aplică numai la liste cu un număr relativ mic de noduri.

O metodă mult mai eficientă este metoda *căutării binare*. Această metodă poate fi aplicată simplu la listele implementate cu ajutorul tabloului de pointeri *tpnod*. În acest scop lista trebuie întîi sortată, de exemplu

crescător, în raport cu valorile cheii. Sortarea se poate realiza procedînd ca în exerciţiul 8.18. în care cuvintele citite de la intrarea standard se păstrează în memoria *heap* şi în paralel cu aceasta se defineşte un tablou cu adresele la care sînt păstrate aceste cuvinte. Apoi se realizează sortarea lor (ordonarea în ordine alfabetică) folosind metoda bulelor.

Pentru o eficienţă mai bună se poate utiliza o altă metodă de sortare (sortare *shell*, sortare rapidă, sortare cu ajutorul arborilor etc.).

11.1.3. Inserarea unui nod într-o listă simplu înlănţuită

Într-o listă simplu înlănţuită se pot face inserări de noduri în diferite poziţii. Amintim cîteva posibilităţi care intervin mai frecvent:

a. inserare înaintea primului nod;

b. inserare înaintea unui nod precizat printr-o cheie;

c. inserare după un nod precizat printr-o cheie;

d. inserare după ultimul nod al listei (adăugarea unui nod nou la listă).

În cele ce urmează se definesc funcţii pentru cazurile a-d indicate mai sus, atît pentru liste simplu înlănţuite, cît şi variantele lor cînd lista se implementează cu ajutorul tabloului de pointeri *tpnod*.

Tipurile TNOD şi TTNOD se consideră declarate ca în paragrafele precedente.

11.1.3.1. Inserarea unui nod într-o listă simplu înlănţuită înaintea primului ei nod

Primul nod al unei liste simplu înlănţuite este nodul spre care nu pointează pointerul *urm* al nici unui nod al listei, adică este nodul care nu este succesorul nici unui nod din listă. Amintim că spre acest nod pointează variabila *prim* dacă lista nu este vidă.

În cazul utilizării tabloului *tpnod*, primul nod al listei este nodul spre care pointează elementul:

tpnod[0]

Funcţia de inserare returnează pointerul spre nodul inserat în listă sau zero în cazul în care nu se poate realiza inserarea. De altfel, toate funcţiile de inserare returnează aceste valori.

```
TNOD *iniprim() /* insereaza nodul curent inaintea primului
                   nod al listei */
{
```

```
 extern TNOD *prim,*ultim;
 TNOD *p;
 int n;
 n=sizeof(TNOD);

/* rezerva memorie heap si incarca datele in zona respectiva */
 if(((p=(TNOD *)malloc(n))!=0)&&(incnod(p)==1)){
     if(prim==0){ /* lista vida */

/* lista se compune numai din nodul curent */
         prim=ultim=p;
         p -> urm=0;  /* nu exista succesor */
     }
     else{
         p -> urm=prim; /* primul nod al listei devine succesorul
                                    nodului care se insereaza */
         prim=p; /* nodul inserat nu este succesorul nici unui alt
                          nod si de aceea prim pointeaza spre el */
     }
     return p;
 }
 if(p==0){
     printf("memorie insuficienta\n");
     exit(1);
 }
/*s-a rezervat memorie pentru nod dar nu s-au incarcat date in nod */
 elibnod(p);
 return 0;
}
```

Inserarea în cazul utilizării tabloului *tpnod* implică eliberarea elementului tpnod[0]. În acest scop se fac atribuirile:

tpnod[i]=tpnod[i-1] pentru i=itpnod, itpnod-1, ..., 2, 1

Se observă că funcţia de inserare înaintea primului nod pentru liste, care foloseşte tabloul de pointeri *tpnod* nu mai este aşa de eficientă ca funcţia *iniprim* definită mai sus.

```
TTNOD *tpiniprim() /* insereaza un nod inaintea primului
                              nod al listei */
{
 extern TTNOD *tpnod[];
 extern int itpnod;
 int n;
```

```
 int i;

 if(itpnod >= MAX){
    printf("lista are prea multe elemente\n");
    return 0;
 }
 n=sizeof(TTNOD);
 if(((p=(TTNOD *)malloc(n))!=0)&&(incnod(p)==1))
 {
/* deplaseaza valorile elementelor tabloului tpnod */
    for(i=itpnod++;i>0;i--)
       tpnod[i]=tpnod[i-1];
    tpnod[0]=p;
    return p;
 }
 if(p==0){
    printf("memorie insuficienta\n");
    exit(1);
 }
 elibnod(p);
 return 0;
}
```

11.1.3.2. Inserarea unui nod într-o listă simplu înlănţuită înaintea unui nod precizat printr-o cheie

Vom presupune că,cheia este de tip *int* ca şi în cazul funcţiei *cnci* definită în paragraful 11.1.2.

```
TNOD *inici(int c)
/* - insereaza un nod inaintea unui nod precizat printr-o cheie numerică;
   - returneaza pointerul spre nodul inserat sau zero daca inserarea
     nu are loc. */
{
 extern TNOD *prim;
 TNOD *q,*q1,*p;
 int n;

 n=sizeof(TNOD);
/* - cauta nodul de cheie=c;
  - la terminarea ciclului:
        q - pointeaza spre nodul respectiv;
       q1 - pointeaza spre nodul precedent celui spre care pointeaza q. */
 q1=0;
```

```
  q=prim;
  while(q){
      if(q -> cheie==c)
          break; /* s-a gasit nodul cautat */
      q1=q;
      q=q -> urm;
  }
  if(q==0){ /* nu exista in lista un nod de cheie=c */
      printf("nu exista in lista un nod de\
             cheie=%d\n",c);
      return 0;
  }

 /* se rezerva zona pentru nod si se incarca datele nodului in zona
    respectiva */
  if(((p=(TNOD *)malloc(n))!=0)&&(incnod(p)==1)){
      if(q==prim){

 /*cheie=c pentru primul nod si inserarea se face inaintea
  primului nod */
          p -> urm=prim;
          prim=p;
      }
      else{

 /*nodul spre care pointeaza p se insereaza intre nodul spre care
   pointeaza q1 si nodul spre care pointeaza q */
          q1-> urm=p; /* succesorul nodului spre care pointeaza
                         q1 este nodul spre care pointeaza p */
          p -> urm=q; /* succesorul nodului spre care pointeaza p
                         este nodul spre care pointeaza q */
      }
      return p;
  }

 /* nu s-a facut inserarea ceruta */
  if(p==0){
      printf("memorie insuficienta\n");
      exit(1);
  }
  elibnod(p);
  return 0;
}
```

În continuare definim funcţia care utilizează tabloul de pointeri *tpnod*.

```
TTNOD *tpinici(int c)
/* - insereaza un nod inaintea unui nod precizat printr-o cheie numerica;
   - returneaza pointerul spre nodul inserat sau zero daca nu are
     loc inserarea. */
{
 extern TTNOD *tpnod;
 extern int itpnod;
 int i,k;

/*daca itpnod >= MAX nu exista loc liber in tabloul de pointeri */
 if(itpnod >= MAX)
    printf("lista are prea multe elemente\n");
    return 0;
 }

/* cauta nodul de cheie=c */
 for(i=0;i<itpnod;i++)
    if(tpnod[i] -> cheie==c)
        break; /* s-a gasit nodul cautat */
 if(i==itpnod){ /* nu exista in lista un nod de cheie=c */
    printf("nu exista in lista un nod de\
          cheie=%d\n",c);
    return 0;
 }
 n=sizeof(TTNOD);

/* rezerva zona pentru nod si se incarca datele nodului in zona
   respectiva */
 if(((p=(TTNOD *)malloc(n))!=0)&&(incnod(p)==1)){

/* deplaseaza valorile elementelor tabloului tpnod:
         tpnod[k]=tpnod[k-1], pentru k=itpnod,itpnod-1,...,i+1. */
    for(k=itpnod++;k>i;k--)
        tpnod[k]=tpnod[k-1];

/* tpnod[i] devine egal cu adresa nodului care se insereaza */
    tpnod[i]=p;
    return p;
 }

/* nu s-a facut inserarea */
 if(p==0){
```

```
        printf("memorie insuficienta\n");
        exit(1)
    }
    elibnod(p);
    return 0;
}
```

11.1.3.3. Inserarea unui nod într-o lista simplu înlănţuită după un nod precizat printr-o cheie

Vom presupune că, cheia este de tip *int*.

```
TNOD *indci(int c)
/* - insereaza un nod dupa un nod precizat printr-o cheie numerica;
   - returneaza pointerul spre nodul inserat sau zero daca inserarea nu
     are loc. */
{
  extern TNOD *prim,*ultim;
  TNOD *p,*q;
  int n;

/* cauta nodul de cheie=c */
  for(q=prim;q;q=q -> urm)
      if(q -> cheie==c)
          break;
  if(q==0){
      printf("nu exista in lista un nod de\
            cheie=%d\n",c);
      return 0;
  }

/* se rezerva zona pentru nod si se incarca datele in nod */
  n=sizeof(TNOD);
  if(((p=(TNOD *)malloc(n))!=0)&&(incnod(p)==1)){

/* se insereaza nodul spre care pointeaza p dupa nodul spre care
   pointeaza q */
      p -> urm=q -> urm; /* nodul succesor nodului spre care
                            pointeaza q devine succesor nodului
                            spre care pointeaza p */
      q -> urm=p; /* succesorul nodului spre care pointeaza q
                     devine nodul spre care pointeaza p */
      if(ultim==q) /* - nodul spre care pointeaza q nu a avut suc-
```

```
        cesor;
      - in acest moment nodul spre care pointeaza p nu are succesor. */
        ultim=p;
      return p;
  }

/* nu s-a reusit inserarea nodului */
  if(p==0){
      printf("memorie insuficienta\n");
      exit(1);
  }
  elibnod(p);
  return 0;
}
```

Funcţia de inserare pentru liste implementate cu ajutorul tabloului de pointeri *tpnod* care realizează inserarea după un nod de cheie dată este asemănătoare cu funcţia *tpinici* definită în paragraful precedent. În acest caz, se modifică instrucţiunea *for* pentru deplasarea elementelor tabloului *tpnod* şi instrucţiunea de atribuire următoare ei:

```
for(k=itpnod++;k>i+1;k--)
    tpnod[k]=tpnod[k-1];
tpnod[i+1]=p;
```

11.1.3.4. Adăugarea unui nod la o listă simplu înlănţuită

Adăugarea unui nod la o lista simplu înlănţuită înseamnă inserarea lui după nodul spre care pointeaza variabila *ultim*. Aceasta înseamnă că după inserarea nodului, variabila *ultim* pointează spre nodul respectiv, acesta devenind nodul din listă care nu are succesor.

```
TNOD *adauga()
/*- adauga un nod la o lista simplu inlantuita;
  - returneaza pointerul spre nodul inserat sau zero daca nu se realizeaza
    inserarea. */
{
  extern TNOD *prim,*ultim;
  TNOD *p;
  int n;

/* se rezerva zona de memorie pentru nod si se incarca datele in zona
   respectiva */
  n=sizeof(TNOD);
  if(((p=(TNOD *)malloc(n))!=0) && (incnod(p)==1)){
```

```
      if(prim==0) /* lista este vida */
          prim=ultim=p;
      else{
          ultim -> urm=p; /* succesorul nodului spre care
                                     pointeaza ultim devine nodul
                                      spre care pointeaza p */
          ultim=p; /* acesta devine nodul spre care
                           pointeaza ultim */
      }
      p -> urm=0; /* nodul spre care pointeaza p nu are succesor */
      return p;
  }

/* nu s-a reusit inserarea nodului in lista */
 if(p==0){
     printf("memorie insuficienta\n");
     exit(1);
 }
 elibnod(p);
 return 0;
}
```

Definim mai jos funcţia de adăugare a unui nod la o listă implementată cu ajutorul tabloului *tpnod*.

```
TTNOD *tpadauga()
/*- adauga un nod la o lista;
  - returneaza pointerul la nodul inserat sau zero daca inserarea
    nu se realizeaza. */
{
 extern TTNOD *tpnod[];
 extern int itpnod;
 int n;

 if(itpnod >= MAX){
     printf("lista are prea multe elemente\n");
     return 0;
 }
 n=sizeof(TTNOD);
 if(((p=TTNOD *)malloc(n))!=0)&&(incnod(p)==1)){
     tpnod[itpnod++]=p;
     return p;
 }
 if(p==0){
```

```
        printf("memorie insuficienta\n");
        exit(1);
    }
    elibnod (p);
    return 0;
}
```

11.1.4. Ştergerea unui nod dintr-o listă simplu înlănţuită

Dintr-o listă simplu înlănţuită se pot şterge noduri. Aceasta se poate realiza în mai multe moduri. În cele ce urmează avem în vedere următoarele cazuri:

a. ştergerea primului nod al listei simplu înlănţuite;

b. ştergerea unui nod precizat printr-o cheie;

c. ştergerea ultimului nod al listei simplu înlănţuite.

Funcţiile de ştergere utilizează funcţia *elibnod*. Această funcţie eliberează zona de memorie alocată nodului care se şterge, precum şi eventualele zone de memorie alocate suplimentar prin intermediul funcţiei *incnod* pentru a păstra diferite componente ale unui nod (de exemplu componente de tip şir de caractere).

Alături de funcţiile obişnuite de ştergere, se definesc astfel de funcţii şi pentru listele implementate cu ajutorul tabloului *tpnod*.

11.1.4.1. Ştergerea primului nod al unei liste simplu înlănţuite

```
void spn() /* sterge primul nod din lista */
{
 extern TNOD *prim,*ultim;
 TNOD *p;

 if(prim==0) /* lista vida */
    return;
 p=prim;
 prim=prim -> urm;
 elibnod(p);
 if(prim==0) /* lista a devenit vida */
    ultim=0;
}
```

În cazul în care lista este implementată cu ajutorul tabloului *tpnod*, se elimină nodul spre care pointează elementul tpnod[0], apoi se deplasează

elementele tabloului *tpnod* folosind atribuirile:

tpnod[i]=tpnod[i+1], pentru i=0, 1, ..., itpnod-2.

```
void tpspn() /* sterge primul nod din lista */
{
 extern TTNOD *tpnod[];
 extern int itpnod;
 int i;

 elibnod(tpnod[0]);
 for(i=0;i<itpnod-1,i++)
    tpnod[i]=tpnod[i+1];
 itpnod--;
}
```

11.1.4.2. Şterge un nod precizat printr-o cheie, dintr-o listă simplu înlănţuită

Vom considera că, cheia este de tip *int*. În acest caz tipul TNOD se defineşte ca în paragraful 11.1.2.

```
void snci(int c) /* sterge nodul pentru care cheie=c */
{
 extern TNOD *prim,*ultim;
 TNOD *q,*q1;

 q1=0;
 q=prim;

/*- se cauta nodul de cheie=c;
  - la terminarea ciclului q pointeaza spre nodul respectiv sau
    q=0 daca nu exista un astfel de nod;
  - q1 pointeaza spre nodul a carui succesor este nodul spre care
    pointeaza q sau q1=0 daca q=prim. */
 while(q){
    if(q -> cheie==c)
       break; /* s-a gasit nodul cautat */
    q1=q;
    q=q -> urm;
 }
 if(q==0){ /* nu exista in lista un nod pentru care cheie=c */
    printf("lista nu contine un nod de\
          cheie=%d\n",c);
    return;
```

```
   }

 /* se sterge nodul spre care pointeaza q */
  if(q==prim){ /* se sterge primul nod din lista */
     prim=prim -> urm;
     elibnod(q);
     if(prim==0)
        ultim=0; /* lista a devenit vida */
  }
  else{
     q1->urm=q->urm; /*succesorul nodului spre care pointeaza
                       q1 devine succesorul nodului spre care
                       pointeaza q */
     if(q==ultim) /* se sterge ultimul nod, deci nodul spre care
                     pointeaza q1 devine ultimul nod al listei */
        ultim=q1;
     elibnod(q);
  }
 }
```

În cazul în care lista este implementată cu ajutorul tabloului *tpnod*, se caută nodul pentru care cheie=c. Fie acesta nodul a cărui adresă este dată de elementul tpnod[i].

După eliminarea nodului respectiv, se realizează atribuirile:

tpnod[k]=tpnod[k+1], pentru k=i, i+1, ..., itpnod-2.

```
void tpsnci(int c) /* sterge nodul de cheie=c */
{
 extern TTNOD *tpnod[];
 extern int itpnod;
 int i;

 for(i=0;i<itpnod;i++)
    if(tpnod[i] -> cheie==c)
       break;
 if(i==itpnod){
    printf("nu exista un nod de cheie=%d\n",c);
    return;
 }
 elibnod(tpnod[i]);
 for(;i<itpnod-1;i++)
    tpnod[i]=tpnod[i+1];
 itpnod--;
```

```
}
```

11.1.4.3. Ştergerea ultimului nod dintr-o listă simplu înlănţuită

```
void sun() /* sterge ultimul nod din lista */
{
 extern TNOD *prim,*ultim;
 TNOD *q,*q1;

 q1=0;
 q=prim;
 if(q==0)
     return; /* lista vida */
 while(q!=ultim){ /* se parcurge lista pina se ajunge la ultimul
                     nod al ei */
     q1=q;
     q=q -> urm;
 }
 if(q==prim) /* - lista contine un singur nod care se sterge;
                - lista devine vida. */
     prim=ultim=0;
 else{

/* - nodul spre care pointeaza q1 are ca succesor nodul spre care
     pointeaza q si acesta este ultimul nod al listei;
   - cum nodul spre care pointeaza q se sterge, nodul spre care pointeaza
     q1 devine ultimul, deci q1 -> urm=0 si ultim=q1. */
     q1 -> urm=0;
     ultim=q1;
 }
 elibnod(q);
}
```

În cazul în care lista se implementează folosind tabloul *tpnod*, operaţia de ştergere a ultimului nod este mult mai eficientă, deoarece nu necesită parcurgerea nodurilor listei.

```
void tpsum() /* sterge ultimul nod al listei */
{
 extern TTNOD *tpnod[];
 extern int itpnod;

 if(itpnod==0)
     return; /* lista vida */
```

```
  elibnod(--itpnod);
}
```

Observaţie:

Funcţiile definite în paragrafele precedente realizează operaţiile cele mai frecvent utilizate asupra listelor. Unele dintre aceste funcţii necesită parcurgerea nodurilor listei, parţial sau în totalitate. Alte funcţii nu necesită astfel de parcurgeri. Funcţiile care parcurg nodurile unei liste au o eficienţă scăzută în comparaţie cu cele care nu fac astfel de parcurgeri. De aceea, se recomandă pe cît posibil utilizarea funcţiilor care nu necesită parcurgerea nodurilor ei. Astfel de funcţii sînt:

iniprim - Inserează nodul curent înaintea primului nod al listei (11.1.3.1.).

adauga - Adaugă un nod la o listă (11.1.3.4.).

spn - Ştergerea primului nod al listei (11.1.4.1).

În cazul listelor implementate cu ajutorul tabloului *tpnod*, cele mai eficiente funcţii sînt cele care nu necesită instrucţiuni ciclice care să se execute asupra elementelor tabloului *tpnod*. Astfel de funcţii sînt:

tpadauga - Adaugă un nod la o listă.

tpsun - Şterge ultimul nod al listei.

De aceste observaţii se ţine, uneori, seama la alegerea modului de implementare a listelor în aplicaţii.

11.1.5. Ştergerea unei liste simplu înlănţuite

Se utilizează funcţia *elibnod* pentru fiecare nod al listei.

```
void streglist() /* sterge o lista simplu inlantuita */
{
 extern TNOD *prim,*ultim;
 TNOD *p;

 while(prim){
     p=prim;
     prim=prim -> urm;
     elibnod(p);
 }
 ultim=0;
}
```

Mai jos definim funcţia de ştergere a listei implementate cu ajutorul tabloului *tpnod*.

```
void tpsterglist() /* sterge lista implementata cu tablou
                      de pointeri */
{
 extern TTNOD *tpnod[];
 extern int itpnod;
 int i;

 for(i=0;i<itpnod;i++)
    elibnod(tpnod[i]);
 itpnod=0;
}
```

Exerciţii:

11.1 Se consideră tipul utilizator:

```
typedef struct tnod{
    char *cuvant;
    int frecventa;
    struct tnod *urm;
}TNOD
```

Se cere să se scrie funcţia *incnod* care încarcă datele curente într-un nod de tip TNOD.

Prin cuvînt se înţelege un şir de litere mici sau mari, ca în exerciţiul 10.48. În acest exerciţiu se defineşte funcţia *citcuv* care citeşte un cuvînt de la intrarea standard şi-l păstrează în memoria *heap*. Funcţia respectivă returnează adresa de început a zonei în care se păstrează cuvîntul citit sau zero în caz că se întîlneşte sfîrşitul de fişier.

Funcţia de faţă apelează funcţia *citcuv* şi atribuie adresa returnată de ea pointerului *cuvant* din nodul curent. De asemenea, se atribuie valoarea 1, variabilei *frecventa*.

Funcţia *incnod* returnează valoarea -1 dacă *citcuv* returnează valoarea zero şi 1 altfel.

FUNCŢIA BXI1

```
int incnod(TNOD *p)
/* incarca datele curente in nodul spre care pointeaza p */
{
```

```
  if((p -> cuvant = citcuv() ) == 0 )
     return -1;
  p ->frecventa = 1;
  return 1;
}
```

11.2 Să se scrie funcţia *elibnod* care eliberează zonele din memoria *heap* ocupate de nodul de tip TNOD definit în exerciţiul precedent.

FUNCŢIA BXI2

```
void elibnod(TNOD *p)
/* elibereaza zonele din memoria heap ocupate de nodul spre
   care pointeaza p */
{
 free( p -> cuvant);
 free(p);
}
```

11.3 Să se scrie funcţia *adauga*, care permite adăugarea unui nod de tip TNOD la o listă simplu înlănţuită, după ultimul nod al listei. Tipul TNOD este cel definit în exerciţiul 11.1.

Să observăm că funcţia *adauga* definită în paragraful 11.1.3.4. este dependentă numai de pointerul *urm* din tipul nodului. De aceea, funcţia *adauga* de mai jos este identică cu cea din paragraful amintit mai sus.

Funcţia de faţă apelează funcţiile *incnod* şi *elibnod* definite în exerciţiile precedente.

FUNCŢIA BXI3

```
TNOD *adauga()
/* - adauga un nod la o lista simplu inlantuita;
   - returneaza pointerul spre nodul adaugat sau zero daca nu s-a
     realizat adaugarea. */
{
 extern TNOD *prim,*ultim;
 TNOD *p;
 int n;

 n = sizeof(TNOD);
 if(((p = (TNOD *)malloc(n)) != 0 ) &&
  (incnod(p) == 1 )) {
```

```
        if(prim == 0 )
            prim = ultim =p;
        else {
            ultim -> urm = p;
            ultim = p;
        }
        p ->urm = 0;
        return p;
    }
    if( p == 0 ) {
        printf("memorie insuficienta\n");
        exit(1);
    }
    elibnod(p);
    return 0;
}
```

11.4 Fie o listă simplu înlănţuită ale cărei noduri au tipul TNOD definit în exerciţiul 11.1.

Să se scrie o funcţie care caută în lista respectivă, nodul pentru care pointerul *cuvant* are ca valoare adresa unui cuvînt dat.

Cu alte cuvinte, pointerul *cuvant* joacă rol de cheie şi se cere să se găsească nodul a cărui cheie pointează spre un cuvînt dat.

Această funcţie este asemănătoare cu funcţia *cnci* definită în paragraful 11.1.2.

FUNCŢIA BXI4

```
TNOD *cncs(char *c)
/* - cauta un nod al listei pentru care cuvintul spre care pointeaza cuvant
     este identic cu cel spre care pointeaza c;
   - returneaza pointerul spre nodul determinat sau zero daca nu exista
     un astfel de nod. */
{
 extern TNOD *prim;
 TNOD *p;

 for(p = prim; p ; p = p->urm)
     if(strcmp(p->cuvant,c) == 0)
         return p;
 return 0;
}
```

11.5 Să se scrie o funcţie care şterge ultimul nod al unei liste simplu înlănţuite ale cărei noduri au tipul TNOD definit în exerciţiul 11.1.

Această funcţie este definită în paragraful 11.1.4.3 şi ea este dependentă numai de pointerul *urm* din tipul nodurilor.

Ea apelează funcţia *elibnod* definită în exerciţiul 11.2.

FUNCŢIA BXI5

```
void sun() /* sterge ultimul nod din lista */
{
 extern TNOD *prim,*ultim;
 TNOD *ql,*q;

 ql = 0;
 if(prim == 0 )
     return;
 for(q = prim; q && q != ultim; q=q->urm)
     ql = q;
 if(q == prim )
     prim = ultim =0;
 else {
     ql -> urm = 0;
     ultim = ql;
 }
 elibnod(q);
}
```

11.6 Să se scrie un program care citeşte cuvintele dintr-un text şi afişează numărul de apariţii al fiecărui cuvînt din textul respectiv.

Cuvîntul se defineşte ca o succesiune de litere mici şi/sau mari. Textul se termină prin sfîrşitul de fişier.

Această problemă a fost rezolvată în exerciţiul 10.50. Problema s-a rezolvat utilizînd un tablou de pointeri ale cărui elemente sînt adrese pentru date păstrate în memoria *heap*. Acest tablou este similar cu tabloul *tpnod* introdus în acest capitol şi utilizat la implementarea listelor.

În exerciţiul de faţă programul se realizează utilizînd o listă simplu înlănţuită ale cărei noduri au tipul TNOD definit în exerciţiul 11.1.

Deşi programul din exerciţiul 10.50 este mai eficient din punct de vedere al timpului de execuţie decît cel de faţă, el are o limitare cu privire la numărul de cuvinte diferite din text.

În exercițiul 10.50 s-a limitat maximul cuvintelor diferite la 100. Evident, această limită poate fi schimbată simplu, modificînd valoarea lui MAX.

În programul de față nu mai este necesar să definim acest maxim. El se execută astfel:

1. La întîlnirea unui cuvînt se construiește un nod pentru cuvîntul respectiv.

   Acesta conține pointerul spre cuvînt, care este păstrat în memoria *heap*; frecvența de apariție a cuvîntului se face egală cu 1.

2. Nodul construit la punctul 1 se adaugă la lista simplu înlănțuită care se construiește.

3. Se caută în listă un nod care să corespundă cuvîntului curent.

   Dacă există un astfel de nod și acesta nu este ultimul nod al listei, atunci se mărește frecvența din nodul respectiv și apoi se șterge ultimul nod al listei (cel adăugat la punctul 2) deoarece cuvîntul există deja în listă.

   După pasul 3 se revine la pasul 1 și ciclul continuă pînă cînd nu mai sînt cuvinte de citit (s-a ajuns la sfîrșitul de fișier). În acest moment se listează cuvintele și frecvența lor de apariție, corespunzătoare nodurilor listei.

PROGRAMUL BXI6

```
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
#include <string.h>

typedef struct tnod {
 char *cuvant;
 int frecventa;
 struct tnod *urm;
} TNOD;

#include "bx48.cpp" /* citcuv  */
#include "bxi1.cpp" /* incnod  */
#include "bxi2.cpp" /* elibnod */
#include "bxi3.cpp" /* adauga  */
#include "bxi4.cpp" /* cncs    */
#include "bxi5.cpp" /* sun     */

TNOD *prim,*ultim;
```

```
main() /* citeste un text si afiseaza frecventa cuvintelor din text */
{
 TNOD *p,*q;

 prim = ultim =0; /* la inceput lista este vida */
 while((p = adauga() ) != 0 )

/* s-a adaugat la lista un nod corespunzator ultimului cuvint citit */
     if((q=cncs(p->cuvant)) != ultim) {

/* - cuvintul exista intr-un nod care nu este ultimul nod al listei;
   - deci exista deja in lista;
   - se mareste frecventa lui si se sterge ultimul nod al listei. */
         q -> frecventa++;
         sun();
     }

/* listeaza cuvintele si frecventa lor */
 for(p = prim; p ; p = p->urm)
     printf("cuvintul: %-51s are frecventa: %d\n",
           p -> cuvant, p -> frecventa);
}
```

11.7 Să se scrie un program care citeşte cuvintele dintr-un text şi scrie numărul de apariţii al fiecărui cuvînt în ordinea alfabetică a cuvintelor respective.

Acest program este asemănător cu cel precedent. Diferenţa constă în faptul că înainte de a lista cuvintele şi frecvenţa lor, se face o ordonare a nodurilor listei în aşa fel încît cuvintele corespunzătoare nodurilor să fie ordonate alfabetic. Aceasta se realizează cu ajutorul funcţiei *ordlist* definită mai jos.

Ordonarea nodurilor se face pe baza schimbării înlănţuirii acestora.

Se utilizează metoda de sortare a bulelor. Potrivit acestei metode, se compară cuvintele corespunzătoare a două noduri vecine. Dacă ele nu sînt în ordine alfabetică, atunci se schimbă ordinea de înlănţuire în listă a nodurilor respective.

PROGRAMUl BXI7

```
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
```

```
#include <string.h>

typedef struct tnod {
 char *cuvant;
 int frecventa;
 struct tnod *urm;
} TNOD;

#include "bx48.cpp" /* citcuv  */
#include "bxi1.cpp" /* incnod */
#include "bxi2.cpp" /* elibnod */
#include "bxi3.cpp" /* adauga */
#include "bxi4.cpp" /* cncs   */
#include "bxi5.cpp" /* sun    */

void ordlist()
/* ordoneaza nodurile listei de tip TNOD in ordinea alfabetica a
   cuvintelor corespunzatoare */
{
 extern TNOD *prim,*ultim;
 TNOD *p,*p1,*q;
 int ind;

 if(prim == 0 )
     return; /* lista vida */
 ind = 1;
 while ( ind ) { /* 1 */

/* cit timp ind nu este zero, trebuie sa se parcurga lista si sa se inverseze
   inlantuirile nodurilor vecine carora le corespund cuvinte care nu sînt
   in ordine alfabetica */
     ind = 0;
     p = prim; /* p pointeaza spre nodul curent */
     p1 = 0; /* p1 pointeaza spre nodul precedent */
     q = p -> urm; /* q pointeaza spre nodul urmator */
     while( q != 0 ) /* exista nod urmator */
         if(stricmp(p->cuvant, q->cuvant) > 0) {

/* - cuvintele corespunzatoare nodului curent si nodului urmator nu sînt
     in ordine alfabetica;
   - se inverseaza inlantuirile lor. */
               if(p == prim) /* nodul curent nu are precedent */
```

```
                prim = q; /* q devine primul nod al listei */
            else
                pl -> urm = q; /* q devine urmatorul lui p1 */
            p -> urm = q -> urm; /* urmatorul lui p devine
                                     urmatorul lui q */
            q -> urm = p; /* urmatorul lui q devine p */
            pl = q; /* precedentul lui p devine q */
            q = p -> urm; /* q devine urmatorul lui p */
            if( q == 0)
                ultim = p; /* p nu mai are urmator */
            ind = 1; /* s-a facut o permutare, deci procesul de
                            ordonare nu este terminat */
        } /* sfirsit if */
        else { /* - nu se schimba inlantuirile nodurilor deoarece
                    cuvintele corespunzatoare sint in ordine alfa-
                    betica;
                  - se avanseaza in lista la perechea urmatoare
                    de noduri. */
            pl = p;
            p = q;
            q = q -> urm;
        } /* sfirsit else */
  } /* sfirsit while 1 */
} /* sfirsit ordlist */

TNOD *prim,*ultim;

main() /* citeste un text si afiseaza frecventa cuvintelor din textul
            respectiv in ordine alfabetica */
{
 TNOD *p;

 prim = ultim = 0;

/* creeaza lista */
 while( adauga() != 0 )
     if((p = cncs(ultim -> cuvant)) != ultim ) {
         p ->frecventa++;
         sun();
     }
```

```
/* sortare */
 ordlist();

/* listare */
 for( p=prim; p; p = p->urm)
     printf("cuvintul: %-51s are frecventa: %d\n",
             p -> cuvant, p -> frecventa);
}
```

11.2. Stive şi cozi

În exerciţiul 7.1 s-a implementat o stivă cu ajutorul unui tablou. În general, o stivă se implementează printr-o listă simplu înlănţuită.

O *stivă* este o listă simplu înlănţuită gestionată conform principiului LIFO (Last in First Out).

Conform acestui principiu, ultimul nod pus în stivă este primul care este scos din stivă.

Stiva, ca şi lista are două capete, *baza* stivei şi *vîrful* stivei.

Asupra unei stive se definesc cîteva operaţii, dintre care cele mai importante sînt:

1. pune un element pe stivă (*push*);
2. scoate un element din stivă (*pop*);
3. şterge (videază) stiva (*clear*).

Primele două operaţii se realizează în vîrful stivei. Astfel, dacă se scoate un element din stivă, atunci acesta este cel din vîrful stivei şi în continuare, cel pus anterior lui pe stivă ajunge în vîrful stivei.

Dacă un element se pune pe stivă, atunci acesta se pune în vîrful stivei.

Pentru a implementa o stivă printr-o listă simplu înlănţuită va trebui să identificăm baza şi vîrful stivei cu capetele listei simplu înlănţuite. Există două posibilităţi:

a. nodul spre care pointează variabila *prim* este baza stivei, iar nodul spre care pointează variabila *ultim* este vîrful stivei;

b. nodul spre care pointează variabila *prim* este vîrful stivei, iar nodul spre care pointează variabila *ultim* este baza stivei.

În cazul *a*, funcţiile *push* şi *pop* se indentifică prin funcţiile *adauga* şi respectiv *sun*, definite în paragrafele 11.1.3.4 şi respectiv 11.1.4.3.

Dacă funcţia *adauga* este eficientă, în schimb funcţia *sun* nu este eficientă.

În cazul *b*, funcţiile *push* şi *pop* se indentifică prin funcţiile *iniprim* şi respectiv *spn*, definite în paragrafele 11.1.3.1. şi respectiv 11.1.4.1.

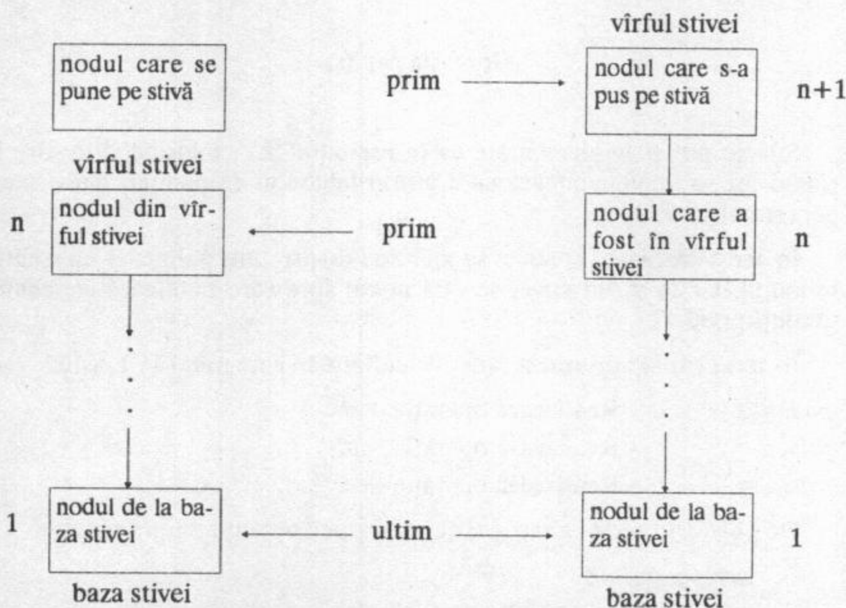
În acest caz ambele funcţii sînt eficiente. De aceea, se recomandă implementarea stivei printr-o listă simplu înlănţuită conform cazului *b* indicat mai sus.

Cele două operaţii *push* şi *pop* pot fi schematizate ca în figura de mai jos.

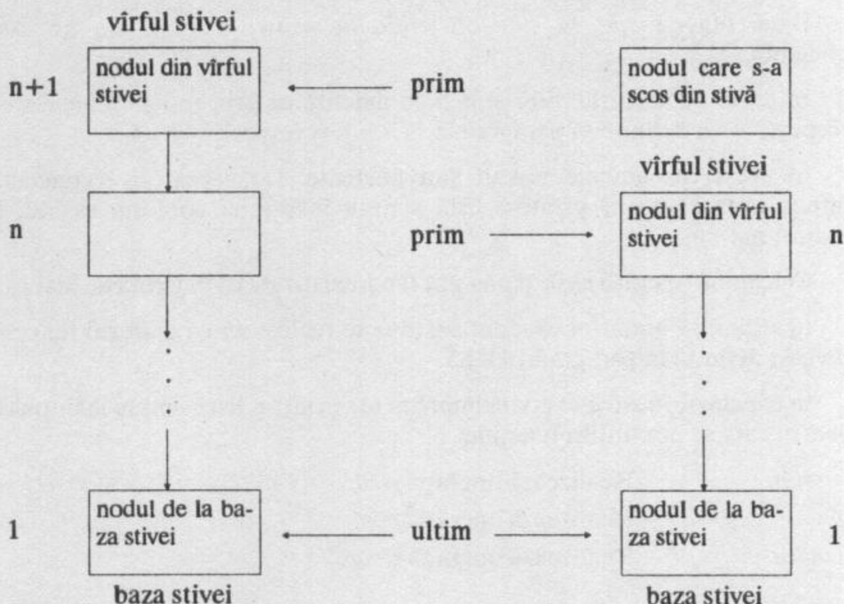
În sfîrşit, să amintim că vidarea stivei se realizează cu ajutorul funcţiei *sterglist* definită în paragraful 11.1.5.

În concluzie, o stivă se poate implementa printr-o listă simplu înlănţuită, pentru care se pot utiliza funcţiile:

iniprim	- Realizează operaţia *push*.
spn	- Realizează operaţia *pop*.
sterglist	- Realizează operaţia *clear*.



Operaţia push (iniprim)



Operaţia pop (spn)

Stivele pot fi implementate ca în exerciţiul 7.1., folosind liste care la rîndul lor se implementează cu ajutorul tabloului de pointeri *tpnod* (vezi paragraful 11.1).

În acest caz, la baza stivei se află nodul spre care pointează elementul tpnod[0], iar în vîrful stivei se află nodul spre care pointează elementul tpnod[itpnod-1].

În acest caz se vor utiliza funcţiile definite în paragraful 11.1, astfel:

tpadauga - Realizează operaţia *push*.

tpsun - Realizează operaţia *pop*.

tpsterglist - Realizează operaţia *clear*.

De obicei, în acest caz se mai definesc două operaţii asupra stivei:

isempty şi *isfull*

Prima se defineşte printr-o funcţie care returnează valoarea 1 dacă stiva este vidă şi zero în caz contrar. Cea de a doua returnează valoarea 1 dacă stiva este plină şi zero în caz contrar. Păstrînd denumirile de mai sus, aceste funcţii se definesc simplu astfel:

```
typedef enum {false,true} Boolean;

Boolean isempty() /* returneaza true daca stiva este vida
                     si false altfel */
{
 extern int itpnod;
 return itpnod==0;
}

Boolean isfull() /* returneaza true daca stiva este plina
                    si false altfel */
{
 extern int itpnod;
 return itpnod >= MAX;
}
```

Un alt principiu de gestiune a listelor simplu înlănţuite este principiul FIFO (First In-First Out).

Conform acestui principiu, primul element introdus în listă este şi primul care este scos din listă.

Despre o listă gestionată în acest fel se spune că formează o *coadă*.

Cele două capete ale listei simplu înlănţuite care implementează o coadă sînt şi capetele cozii. Asupra cozilor se definesc trei operaţii, ca şi asupra stivelor:

a. pune un element în coadă;

b. scoate un element din coadă;

c. ştergerea (vidarea) unei cozi.

Pentru a respecta principiul FIFO, vom pune un element în coadă folosind funcţia *adauga* şi vom scoate un element din coadă folosind funcţia *spn*. Deci, la un capăt al cozii se pun elemente în coadă, iar din celălalt capăt se scot elementele din coadă.

Ambele funcţii, *adauga* şi *spn* sînt funcţii eficiente.

Ştergerea unei liste se realizează cu ajutorul funcţiei *sterglist*.

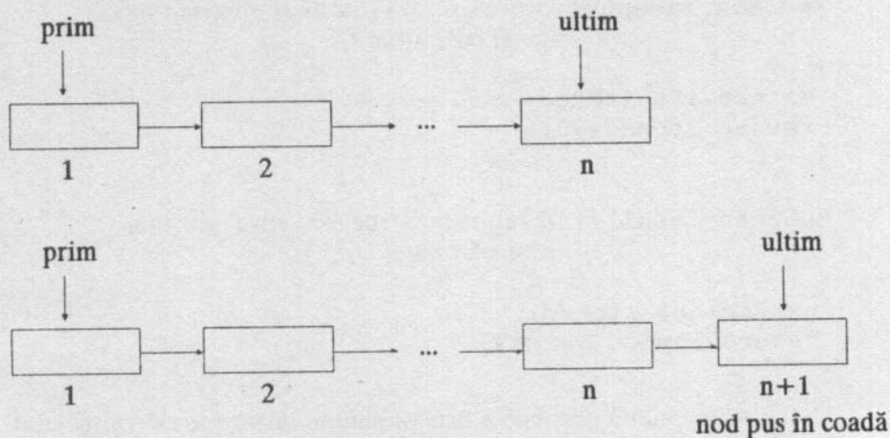
Primele două funcţii sînt schematizate în figura următoare.

În concluzie, *coada* este o listă simplu înlănţuită pentru care se pot utiliza funcţiile:

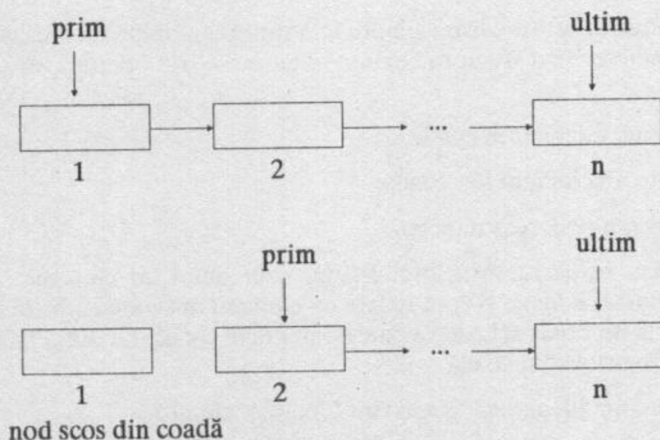
adauga - Realizează operaţia de adăugare a unui nod la coadă.

spn - Realizează operaţia de scoatere a unui nod din coadă.

sterglist - Realizează ştergerea nodurilor existente în coadă.



Operaţia de punere în coadă



Operaţia de scoatere din coadă

Cozile definite ca mai sus corespund celor din viaţa de toate zilele şi din această cauză ele se folosesc frecvent în probleme de simulare a fenomenelor reale.

Stivele se utilizează la descrierea proceselor recursive, inversarea ordinii elementelor unei mulţimi ordonate etc.

Exerciţii:

11.8 Într-o gară se consideră un tren de marfă ale cărui vagoane sînt inventariate într-o listă, în ordinea vagoanelor. Lista conţine, pentru fiecare vagon, următoarele date:

- codul vagonului (9 cifre);
- codul conţinutului vagonului (9 cifre);
- adresa expeditorului (4 cifre);
- adresa destinatarului (4 cifre).

Deoarece în gară se inversează poziţia vagoanelor, se cere listarea datelor despre vagoanele respective în noua lor ordine.

În acest scop se crează o stivă în care se păstrează datele fiecărui vagon. Datele corespunzătoare unui vagon constituie un element al stivei, adică un nod al listei simplu înlănţuite. După ce datele au fost puse pe stivă, ele se scot de acolo şi se listează. În acest mod se obţine lista vagoanelor în ordine inversă celei iniţiale.

Stiva este o listă simplu înlănţuită pe care o gestionăm folosind funcţiile:

iniprim şi *spn*.

Aceste funcţii sînt generale şi ele sînt definite în paragrafele 11.1.3.1. şi respectiv 11.1.4.1.

Funcţia *iniprim* apelează funcţia *incnod* şi *elibnod*, iar funcţia *spn* numai funcţia *elibnod*. Aceste două funcţii (*incnod* şi *elibnod*) sînt specifice aplicaţiei.

Funcţia principală apelează repetat funcţia *iniprim* pentru a pune datele pe stivă, apoi le scoate de pe stivă şi le listează, pînă cînd aceasta devine vidă.

PROGRAMUL BXI8

```
#include <stdio.h>
#include <alloc.h>
#include <stdlib.h>

typedef struct tnod {
 long cvag;
 long cmarfa;
 int exp;
```

```
 int dest;
 struct tnod *urm;
} TNOD;

#include "bviii2.cpp" /* pcit_int    */
#include "bviii3.cpp" /* pcit_int_lim */

int incnod(TNOD *p) /* incarca un nod cu datele despre vagoane */
{
 char t[255];
 char er[] = "s-a tastat EOF in pozitie rea\n";
 long cod;
 int icod;

/* citeste cod vagon */
 for ( ; ; ) {
     printf("cod vagon: ");
     if(gets(t) == 0 )
         return -1; /* nu mai sînt date */
     if(sscanf(t,"%ld",&cod)==1 && cod >= 0 &&
           cod <= 999999999)
         break;
     printf("cod vagon eronat\n");
 }
 p -> cvag = cod;

/* citeste cod marfa */
 for( ; ; ) {
     printf("cod marfa: ");
     if(gets(t) == 0 ) {
         printf(er);
         return 0;
     }
     if(sscanf(t,"%ld",&cod)==1 && cod >= 0 &&
           cod <= 999999999)
     break;
     printf("cod marfa eronat\n");
 }
 p -> cmarfa = cod;

/* citeste cod expeditor */
 if(pcit_int_lim("cod expeditor: ",0,9999,
        &icod) == 0 ) {
```

```
     printf(er);
     return 0;
 }
 p -> exp = icod;

/* citeste cod destinatar */
 if(pcit_int_lim("cod destinatar: ",0,9999,
        &icod) == 0 ) {
     printf(er);
     return 0;
 }
 p -> dest = icod;
 return 1;
} /* sfirsit incnod */

void elibnod(TNOD *p) /* elibereaza nodul spre care
                         pointeaza p */
{
 free(p);
} /* sfirsit elibnod */

TNOD *iniprim() /* insereaza nodul curent inaintea primului nod
                   al listei - push */
{
 extern TNOD *prim,*ultim;
 TNOD *p;
 int n;

 n = sizeof(TNOD);
 if(((p = (TNOD *)malloc(n)) != 0 ) &&
        (incnod(p) == 1 )) {
    if(prim == 0 ) {
        prim = ultim = p;
        p -> urm =0;
    }
    else {
        p -> urm = prim;
        prim = p;
    }
    return p;
 }
 if(p == 0 ) {
    printf("memorie insuficienta\n");
```

```
        exit(1);
    }
    elibnod(p);
    return 0;
   } /* sfirsit iniprim */

   void spn() /* sterge primul nod din lista - pop */
   {
    extern TNOD *prim, *ultim;
    TNOD *p;

    if(prim == 0 )
        return ;
    p = prim;
    prim = prim -> urm;
    elibnod(p);
    if(prim == 0 )
        ultim = 0;
   } /* sfirsit spn */

   TNOD *prim, *ultim;

   main() /* listeaza inventarul vagoanelor in ordinea inversa citirii lor */
   {
    prim = ultim =0; /* la inceput stiva este vida */

   /* se creaza stiva apelind iniprim pina cind aceasta returneaza valoarea
      zero */
    while(iniprim() != 0)
        ;

   /* - se listeaza elementul din virful stivei si apoi se sterge din stiva;
        - se repeta pina cind stiva devine vida. */
    while(prim != 0 ) {
        printf("cod vagon: %ld\tcontinut: %ld\n",
               prim -> cvag, prim -> cmarfa);
        printf("expeditor: %d\tdestinatar: %d\n",
               prim -> exp, prim -> dest);
        spn();
    }
   } /* sfirsit main */
```

11.9 Cozile se folosesc adesea la simularea fenomenelor reale pe calculator.

Un exemplu simplu de simulare cu ajutorul cozilor este descris mai jos.

Presupunem o agenţie C.E.C. mică cu un singur ghişeu, care se deschide la ora 8. Agenţia se închide la ora 16, dar publicul aflat la coadă este deservit în continuare. Cu alte cuvinte, după ora 16 publicul nu mai are voie să intre în agenţie ca să se aşeze la coada de la ghişeu. Întrucît s-a constatat că de obicei coada este destul de mare la ora închiderii, se ridică problema oportunităţii deschiderii a încă unui ghişeu la agenţia respectivă. În acest scop se realizează o simulare pe calculator a situaţiei existente care să stabilească o medie a orelor suplimentare efectuate zilnic pe o perioadă de un an.

Pentru a simula acest fenomen se au în vedere operaţiile efectuate la ghişeu, după cum urmează:

Operaţie	**Timp de execuţie**
1 - depunere	5 minute
2 - restituire fără confirmare	7 minute
3 - depunere pe un carnet nou	10 minute
4 - restituire cu confirmare	20 minute
5 - lichidare	25 minute

Operaţiile care se efectuează nu sînt uniform repartizate. În medie, operaţiile 1 şi 2 se execută cel mai frecvent iar operaţia 5 cel mai rar. În medie, s-a constatat că operaţiile 3 şi 4 se solicită triplu faţă de operaţia 5, iar operaţiile 1 şi 2 sînt de 7 ori mai solicitate decît operaţia 5.

Avînd în vedere acest fapt, considerăm o metodă de simulare a operaţiilor care se execută la ghişeu bazată pe numere pseudoaleatoare. Dacă se ia în seamă operaţia 5, ca unitate, înseamnă că pentru operaţiile 1 şi 2 vom alege ponderea 7, iar pentru operaţiile 3 şi 4 ponderea 3:

Operaţie	**Pondere**
operaţia 1	7
operaţia 2	7
operaţia 3	3
operaţia 4	3
operaţia 5	1
Total	21

Se vor genera numere pseudoaleatoare situate în intervalul [1,21], iar operaţia se defineşte astfel:

Fie *r* un număr pseudoaleator din intervalul [1,21]:

- dacă 1 <= r <= 7, atunci se realizează operaţia 1;
- dacă 8 <= r <= 14, atunci se realizează operaţia 2;
- dacă 15 <= r <= 17, atunci se realizează operaţia 3;
- dacă 18 <= r <= 20, atunci se realizează operaţia 4;
- dacă r = 21, atunci se realizează operaţia 5.

Numerele pseudoaleatoare pot fi generate folosind funcţiile *random* şi *srand* din biblioteca limbajelor C şi C++.

Funcţia *random* are prototipul:

int random(int *n*);

Ea returnează un număr pseudoaleator aflat în intervalul [0,n) (număr natural mai mic decît *n*).

Numerele pseudoaleatoare se generează printr-un proces de calcul iterativ. Acest proces porneşte cu o valoarea iniţială care poate fi definită de programator apelînd funcţia *srand*. Această valoare iniţială se numeşte *sămînţa* şirului de numere pseudoaleatoare.

Funcţia *srand* are prototipul:

void srand(unsigned *n*);

unde:

n - Este valoarea la care se setează sămînţa după apelul funcţiei *srand*.

Ambele funcţii au prototipul în fişierul *stdlib.h*.

Programul de simulare a problemei indicate mai sus construieşte o coadă de aşteptare cu persoanele care sosesc la agenţie în intervalul de timp indicat mai sus.

Apelînd funcţia *random* se determină operaţia solicitată de fiecare persoană aflată la coadă. Se scot elementele din coadă respectînd principiul FIFO şi procesul continuă pînă la ora 16 cînd se sistează operaţia de adăugare. Se determină timpul necesar pentru realizarea operaţiilor solicitate de persoanele aflate la coadă.

Rămîne de precizat modul în care vin persoanele la agenţie. Vom presupune că intervalul de timp între două persoane care vin la agenţie este aleator şi că acesta este de maximum 15 minute. În acest caz se poate apela

funcţia *random* cu parametrul 15 şi valoarea:

random(15)+1

va reprezenta intervalul la care soseşte persoana următoare la agenţie.

Programul foloseşte o variabilă globală *timpcrt* a cărei valoare este numărul minutelor scurse de la ora 8 şi pînă în momentul în care a sosit ultima persoană la agenţie.

Avem 8 ore de lucru la ghişeu, deci timpcrt<=8*60=480 minute.

O altă variabilă care numără minutele rezultate din deservirea persoanelor care s-au aflat la coadă la ghişeu este *timpghiseu*.

Valorile acestor variabile satisfac relaţia:

timpghiseu <= timpcrt

Cînd o persoană se aşează la coadă, atunci *timpcrt* se măreşte cu timpul dat de expresia:

random(15)+1.

Cînd o persoană se scoate din coadă (a fost deservită), *timpghiseu* se măreşte cu timpul necesar pentru operaţia solicitată de persoana respectivă, operaţie care se defineşte cu ajutorul expresiei:

random(21)+1.

Deoarece *timpcrt* defineşte timpul curent (numărul de minute care s-au socotit începînd cu ora 8 şi pînă în momentul în care ultima persoană s-a aşezat la coadă), rezultă că persoana din faţă este deservită (scoasă din coadă) numai dacă:

timpghiseu+timp necesar pentru operaţia ei <= timpcrt

Altfel se adaugă o nouă persoană la coadă dacă timpcrt<=480.

De asemenea, persoana din faţa cozii este deservită, dacă timpcrt > 480.

PROGRAMUL BXI9

```
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
#include <conio.h>

typedef struct tnod {
 int timpoperatie;
 struct tnod *urm;
```

```
} TNOD;

#define MAXTIMP 480
#define INTERVAL 15

int incnod(TNOD *p)
/* - incarca nodul curent daca timpul curent nu depaseste pe cel admis si
     returneaza 1;
   - altfel returneaza -1. */
{
 extern int timpcrt;
 int r;

/* determina timpul, in minute, la care soseste persoana la agentie */
 timpcrt += random(INTERVAL) + 1;
 if(timpcrt > MAXTIMP )
     return -1; /* agentie inchisa */

/* determina operatia si pastreaza in nod timpul necesar operatiei
    respective */
 r = random(21) + 1;
 if(r <= 7 )
     r = 5;
 else
     if( r <= 14 )
         r = 7;
     else
         if( r <= 17 )
             r = 10;
         else
             if( r <= 20 )
                 r = 20;
             else
                 r = 25;
 p -> timpoperatie = r;
 return 1;
} /* sfirsit incnod */

void elibnod( TNOD *p)
/* elibereaza zona ocupata de nodul listei spre care pointeaza p */
{
 free(p);
} /* sfirsit elibnod */
```

```
#include "bxi3.cpp" /* adauga */

void spn() /* sterge primul nod al listei(cozii) */
{
 extern TNOD *prim,*ultim;
 TNOD *p;

 if( prim == 0 )
     return;
 p = prim;
 prim = prim -> urm;
 elibnod(p);
 if(prim == 0 )
     ultim = 0;
} /* sfirsit spn */

TNOD *prim,*ultim;
int timpcrt;

main()
/* simuleaza cozile de la o agentie CEC pe o perioada de 1 an */
{
 int timpghiseu;
 int i;

 for(i=1;i <= 360;i++){
     srand(i*10); /* seteaza saminta sirului pseudoaleator */

/* initializari la deschiderea agentiei */
     timpcrt = timpghiseu= 0;
     prim = ultim = 0; /* coada este vida */
     while( adauga() )   /* ultima persoana sosita la agentie
                            se pune la coada */

/* - se deservesc persoanele aflate la coada;
   - trebuie ca:
      1 - prim != 0 (altfel nu exista coada);
      2 - timpghiseu <= timpcrt. */
         while ( prim != 0 &&
                    (timpghiseu <= timpcrt) ) {

/* se deserveste prima persoana din coada */
             timpghiseu += prim -> timpoperatie;
```

```
/* se elimina din coada persoana deservita */
                spn();
            }

/* - se inchide agentia;
   - se deservesc in continuare persoanele aflate la coada la ghiseu. */
     while( prim != 0) {
          timpghiseu += prim -> timpoperatie;
          spn();
     }

/* se afiseaza timpul suplimentar in minute */
     if(timpghiseu - 480 > 0){
          printf("timp peste ora 16: %d minute\n",
                    timpghiseu - 480);

          if(i%22==0){
               printf("Actionati o tasta pentru a\
                         continua\n");
               getch();
          }
     }
   }
} /* sfirsit main */
```

11.3. Listă circulară simplu înlănţuită

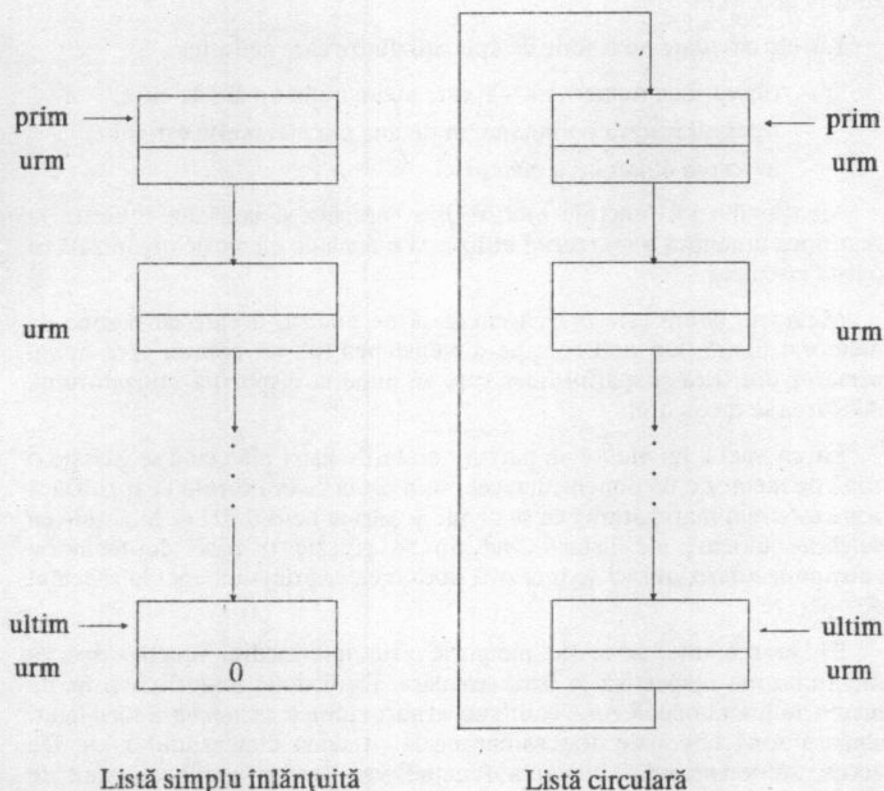
În paragraful 11.1 s-a definit lista simplu înlănţuită ca o mulţime ordonată de noduri, fiecare nod conţinînd un pointer spre un alt nod al listei, numit următorul nodului respectiv, exceptînd un singur nod, care nu mai are următor. Acest nod, care nu mai are următor constituie un capăt al listei simplu înlănţuite. Tot în paragraful amintim mai sus, am convenit ca spre acest nod să pointeze variabila *ultim*. De asemenea, lista simplu înlănţuită conţine un nod care nu este următorul nici unui alt nod al ei. Acest nod constituie celălalt capăt al listei şi spre el pointează variabila *prim*.

Pointerul prezent în fiecare nod al listei care defineşte ordinea nodurilor a fost numit *urm*. Conform celor spuse mai sus, ultim -> urm=0. Dacă într-o listă simplu înlănţuită schimbăm valoarea expresiei ultim -> urm făcînd:

ultim -> urm=prim

atunci lista simplu înlănţuită devine o *listă simplu înlănţuită circulară*. În continuare prin *listă circulară* vom înţelege o listă circulară simplu înlănţuită.

Procesul de transformare a listei simplu înlănţuite în listă circulară este schematizat în figura de mai jos.



Transformarea unei liste simplu înlănţuită în listă circulară

Într-o listă circulară toate nodurile sînt echivalente: fieeare nod are un următor şi fiecare nod este următorul unui nod. Într-o astfel de listă nu mai sînt capete şi de aceea nu mai sînt necesare variabilele *prim* şi *ultim*. Gestiunea nodurilor listei circulare se realizează folosind o variabilă globală care pointează spre un nod oarecare al listei. Numim *ptrnod* această variabilă. Ea se declară astfel:

TNOD *ptrnod;

unde:

TNOD - Este tipul comun nodurilor listei.

Asupra listelor circulare se definesc aceleaşi operaţii ca şi asupra listelor simplu înlănţuite.

Listele circulare au o serie de aplicaţii dintre care amintim:

- operaţii cu numere întregi care au un număr mare de cifre;
- operaţii asupra polinoamelor de una sau mai multe variabile;
- alocarea dinamică a memoriei.

Menţionăm că funcţiile *malloc*, *free*, precum şi celelalte utilizate la gestiunea dinamică a memoriei utilizează o zonă de memorie organizată ca o listă circulară.

Memoria liberă este o listă circulară de noduri, fiecare cu o zonă de memorie liberă. Un nod conţine dimensiunea lui, un pointer spre nodul următor din listă şi spaţiul liber care se pune la dispoziţia utilizatorului. Alocarea se face astfel:

La un apel a lui *malloc* se parcurg nodurile listei pînă cînd se găseşte o zonă de memorie de dimensiune cel puţin egală cu cea cerută la apel. Dacă zona este mai mare, atunci ea se divide şi partea neutilizată se înlănţuie cu celelalte blocuri ale listei. Cînd nu se găseşte o zonă de memorie corespunzătoare, atunci se încearcă obţinerea ei printr-un apel la sistemul de operare.

Eliberarea unei zone de memorie prin intermediul funcţiei *free* va înlănţui zona respectivă la lista circulară. Dacă două noduri cu zone de memorie liberă ocupă zone contigue, atunci ele se concatenează formînd o singură zonă liberă de dimensiune egală cu suma dimensiunilor lor. De aceea, se recomandă utilizarea funcţiei *free* pentru a elibera zone de memorie de îndată ce nu mai este nevoie de ele. În felul acesta se poate preîntîmpina divizarea excesivă a zonei gestionată dinamic prin funcţiile de felul lui *malloc* şi *free*.

Mai jos definim funcţii pentru a realiza operaţiile de bază asupra listelor circulare.

11.3.1. Crearea unei liste circulare

La crearea unei liste circulare, ca şi la crearea unei liste simplu înlănţuite, se utilizează funcţiile *incnod* şi *elibnod*.

Prima se apelează pentru a încărca datele curente într-un nod al listei, iar cea de a doua pentru a elibera zonele de memorie alocate pentru un nod.

Amintim că funcţia *incnod* returnează:

0 - La eroare.

1 - La încărcarea normală.

-1 - Nu mai sînt date de încărcat în nod.

Funcţia de creare a listei circulare returnează:

0 - La eroare.

-1 - La crearea fără erori a listei.

```
int ccrelist() /* - creaza o lista circulara;
                  - returneaza:
                     0 - la eroare;
                    -1 - la creare normala. */
{
 extern TNOD *ptrnod;
 int i,n;
 TNOD *p;

 n=sizeof(TNOD);
 ptrnod=0; /* lista este vida la inceput */
 while(((p=(TNOD *)malloc(n))!=0)&&
          ((i=incnod(p))==1))

/* s-a rezervat zona pentru nod si s-au incarcat date in zona respectiva */
     if(ptrnod==0){ /* lista vida */
         ptrnod=p;
         ptrnod -> urm=p;
     }
     else{ /* nodul curent se insereaza dupa cel spre care pointeaza
               ptrnod */
         p -> urm=ptrnod -> urm;
         ptrnod -> urm=p;
         ptrnod=p; /* ptrnod pointeaza spre ultimul nod inserat in
                      lista */
     } /* sfirsit else */

/* sfirsit while: p=0 sau incnod nu a returnat valoarea 1 */
 if(p==0){ /* nu s-a rezervat zona pentru nod */
```

```
        printf("memorie insuficienta\n");
        exit(1);
    }
    /* - s-a rezervat zona pentru nod dar incnod nu a reusit sa incarce date in
       zona respectiva returnind o valoare diferita de unu, deci zero sau -1;
      - valoarea returnata de incnod va fi returnata si de functia
       ccrelist. */

    elibnod(p); /*elibereaza zona de memorie rezervata pentru nod
                  si care n-a mai fost incarcata cu date */
    return i; /* i are ca valoare valoarea returnata de incnod */
}
```

Observaţie:

Funcţia *ccrelist* a fost scrisă mai compact decît funcţia *crelist* utilizînd expresia:

((p=(TNOD *)malloc(n))!=0)&&((i=incnod(p))==1)

care se evaluează astfel:

Se apelează funcţia *malloc* pentru a rezerva *n* octeţi în memoria *heap*.

În cazul în care se poate rezerva o zonă de *n* octeţi, lui *p* i se atribuie o valoare diferită de zero şi deci primul operand al operatorului && are valoarea adevărat. În acest caz se evaluează operandul al doilea al operatorului &&, care apelează funcţia *incnod* pentru a încărca datele curente în zona a cărei adresă de început este valoarea lui *p*.

În cazul în care nu se pot rezerva *n* octeţi în memoria *heap*, lui *p* i se atribuie valoarea zero şi deci primul operand al operatorului && este fals.

În acest caz întreaga expresie este falsă şi deci nu se mai evaluează cel de al doilea operand al operatorului &&.

De asemenea, expresia este falsă şi în cazul în care lui *p* i s-a atribuit o valoare diferită de zero, dar *incnod* a returnat o valoare diferită de unu.

11.3.2. Accesul la un nod al unei liste circulare

Ca în cazul listelor simplu înlănţuite şi în cazul listelor circulare putem căuta un nod după o cheie sau mai multe chei. În cazul listelor circulare, căutarea va începe cu nodul spre care pointează variabila globală *ptrnod*.

Mai jos, se defineşte funcţia *ccnci* care este analogă funcţiei *cnci* definită

în cazul listelor simplu înlănţuite. Ea caută un nod după o cheie numerică şi returnează una din valorile:

- pointerul spre nodul căutat;
- 0 dacă nu există un astfel de nod.

Amintim că în acest caz tipul nodurilor listei se declară astfel:

typedef struct *tnod* {
    declaratii
    int *cheie*;
    declaratii
} *TNOD*;

```
TNOD *ccnci(int c)
/* - cauta nodul de cheie=c;
   - returneaza pointerul spre nodul respectiv sau 0 daca nu exista
     un astfel de nod. */
{
 extern TNOD *ptrnod;
 TNOD *p;

 p=ptrnod;
 if(p==0) /* lista vida */
    return 0;
 do{
    if(p -> cheie==c)
       return p;
    p=p -> urm;
 }while(p!=ptrnod);
 return 0;
}
```

11.3.3. Inserarea unui nod într-o listă circulară

Considerăm două operaţii de inserare a unui nod într-o listă simplu înlănţuită:

- inserarea unui nod înaintea unuia precizat printr-o cheie numerică de tip *int*;
- inserarea unui nod după unul precizat printr-o cheie numerică de tip *int*.

Cititorul poate defini şi alte funcţii de inserare similare cu cele prezentate mai jos.

11.3.3.1. Inserarea unui nod înaintea unuia precizat printr-o cheie de tip int

Nodurile listei au tipul TNOD indicat în paragraful 11.3.2.

```
TNOD *cinici(int c)
/* - inserarea unui nod intr-o lista circulara inaintea unui nod precizat
     printr-o cheie de tip int;
   - returneaza pointerul spre nodul inserat sau zero daca inserarea nu
     are loc. */
{
 extern TNOD *ptrnod;
 TNOD *p,*q,*q1;
 int n;

 if(ptrnod==0)
     return 0; /* lista vida */
 q=ptrnod;
 do{
     q1=q;
     q=q -> urm;
     if(q -> cheie==c)
         break; /* s-a gasit nodul inaintea caruia se va face
                   inserarea */
 }while(q!=ptrnod);
 if(q -> cheie!=c){
     printf("nu exista un nod de cheie=%d\n",c);
     return 0; /* nu s-a facut nici o inserare */
 }

/* rezerva zona pentru nod si incarca datele in nodul respectiv */
 n=sizeof(TNOD);
 if(((p=(TNOD *)malloc(n))!=0)&&(incnod(p)==1)){
     q1 -> urm=p;
     p -> urm=q;
     return p;
 }
 if(p==0){
     printf("memorie insuficienta\n");
     exit(1);
 }
 elibnod(p);
 return 0;
}
```

11.3.3.2. Inserarea unui nod după un nod precizat printr-o cheie de tip int

Tipul TNOD se defineşte ca în paragraful 11.3.2.

```
TNOD *cindci(int c)
/* insereaza un nod intr-o lista circulara dupa un nod precizat printr-o
   cheie de tip int */
{
 extern TNOD *ptrnod;
 TNOD *p,*q;
 int n;

 if(ptrnod==0)
     return 0; /* lista vida */
 q=ptrnod;
 do{
     if(q -> cheie==c)
         break;
     q=q -> urm
 }while(q!=ptrnod);
 if(q -> cheie !=c){
     printf("nu exista un nod de cheie =%d\n",c);
     return 0;
 }

/* se face inserarea dupa nodul spre care pointeaza q */
 n=sizeof(TNOD);
 if(((p=(TNOD *)malloc(n))!=0)&&(incnod(p)==1)){
     p -> urm=q -> urm;
     q -> urm=p;
     return p;
 }
 if(p==0){
     printf("memorie insuficienta\n");
     exit(1);
 }
 elibnod(p);
 return 0;
}
```

11.3.4. Ştergerea unui nod dintr-o listă circulară

Dăm mai jos o funcţie care permite ştergerea dintr-o listă circulară a unui nod precizat printr-o cheie de tip *int*.

În cazul în care variabila *ptrnod* pointează chiar spre nodul care se şterge, convenim ca *ptrnod* să pointeze spre nodul precedent celui şters, dacă lista n-a devenit vidă. În acest ultim caz, lui *ptrnod* i se atribuie valoarea zero.

Tipul TNOD se defineşte ca în cazul paragrafului precedent.

După modelul funcţiei de mai jos se pot defini şi alte funcţii pentru a şterge noduri dintr-o listă circulară.

```
void csnci(int c) /* sterge nodul pentru care cheie=c */
{
 extern TNOD *ptrnod;
 TNOD *p,*p1;

 if(ptrnod==0)
     return; /* lista vida */
 p=ptrnod;
 do{
     p1=p;
     p=p -> urm;
     if(p->cheie==c)
         break;
 }while(p!=ptrnod);
 if(p -> cheie!=c){
     printf("lista nu contine un nod de\
           cheie=%d\n",c);
     return;
 }
 if(p==p -> urm){ /* lista are un singur nod */
     ptrnod=0;
 }
 else{
     p1 -> urm=p ->urm;
     if(p==ptrnod) /* se sterge nodul spre care pointeaza
                      ptrnod */
         ptrnod=p1;
 }
 elibnod(p);
}
```

11.3.5. Ştergerea unei liste circulare

```
void csterglist() /* sterge o lista circulara */
{
 extern TNOD *ptrnod;
 TNOD *p,*p1;

 if((p=ptrnod)==0)
     return; /* lista vida */
 do{
     p1=p;
     p=p -> urm;
     elibnod(p1);
 }while(p!=ptrnod);
 ptrnod=0;
}
```

Exerciţii:

11.10 Se consideră tipul TNOD declarat ca mai jos:

```
typedef struct tnod {
    char *cuv;
    struct tnod *urm;
} TNOD;
```

Acest tip se utilizează în toate exerciţiile de la acest paragraf.

Mai jos, definim funcţia care încarcă datele într-un nod de tipul TNOD.

FUNCŢIA BXI10

```
int incnod(TNOD *p)
/*- incarca datele in nodul spre care pointeaza p;
  - returneaza:
       -1 la intilnirea sfirsitului de fisier;
        1 altfel. */
{
 char t[255];

 p -> cuv = 0;/* initializarea cu pointerul nul */
 printf("tastati pe un rind cuvintul curent\n");
 if(gets(t) == 0 )
     return -1; /* s-a tastat EOF */
```

```
/* rezerva zona pentru rindul citit */
 if((p -> cuv =
      (char *)malloc(strlen(t) + 1)) == 0 ) {
    printf("memorie insuficienta\n");
    exit(1);
 }

/* pastreaza rindul citit in memoria heap */
 strcpy(p -> cuv , t);
 return 1;
}
```

11.11 Să se definească funcţia *elibnod* care eliberează zonele de memorie ocupate de un nod de tip TNOD.

FUNCŢIA BXI11

```
void elibnod(TNOD *p)
/* elibereaza zonele de memorie ocupate de nodul spre care
  pointeaza p */
{
 free( p -> cuv);
 free(p);
}
```

11.12 Să se scrie funcţia *ccrelist* care crează o listă circulară ale cărei noduri sînt de tipul TNOD definit în exerciţiul 11.10.

FUNCŢIA BXI12

```
int ccrelist()
/* - creaza o lista circulara;
   - returneaza:
        0 la eroare;
       -1 altfel. */
{
 extern TNOD *ptrnod;
 int i,n;
 TNOD *p;

 n = sizeof(TNOD);
 ptrnod = 0;
 while(((p = (TNOD *)malloc(n)) !=0 ) &&
```

```
    ((i = incnod(p)) == 1 ))
      if(ptrnod == 0 ) {
          ptrnod = p;
          ptrnod -> urm = p;
      }
      else {
          p -> urm = ptrnod -> urm;
          ptrnod -> urm = p;
          ptrnod = p;
      }
  if( p == 0 ) {
      printf("memorie insuficienta\n");
      exit(1);
  }
  elibnod(p);
  return i;
}
```

11.13 Să se scrie o funcţie care caută un nod al listei circulare create prin funcţia *ccrelist*, nod pentru care pointerul *cuv* pointează spre un şir de caractere dat.

Funcţia returnează pointerul spre nodul respectiv sau zero dacă nu există un astfel de nod.

FUNCŢIA BXI13

```
TNOD *ccncs( char *c)
/* - cauta nodul pentru care cuv si c pointeaza spre
    acelasi sir de caractere;
  - returneaza:
        pointerul spre nodul respectiv sau zero daca nu exista un
        astfel de nod. */
{
 extern TNOD *ptrnod;
 TNOD *p;

 p = ptrnod;
 if(ptrnod == 0)
     return 0; /* lista vida */
 do {
     if(strcmp( p -> cuv, c) == 0)
         return p;
     p = p -> urm;
```

```
  } while ( p != ptrnod);
  return 0;
}
```

11.14 Fie lista circulară creată cu ajutorul funcţiei *ccrelist* definită în exerciţiul 11.12., fie *pnod* pointerul spre un nod al listei circulare pentru care:

pnod -> cuv

pointează spre un şir dat şi n > 1 un întreg de tip *int*. Se cere nodul din listă obţinut în urma eliminării nodurilor din listă în felul următor:

1. Se porneşte cu nodul imediat următor nodului care conţine pointerul spre şirul dat şi se elimină din listă al n-lea nod care urmează după acest nod.
2. Se execută pasul 1 continuînd cu nodul imediat următor celui şters, pînă cînd lista se reduce la un singur nod.

Nodul la care s-a redus lista este cel căutat.

Această problemă se dă adesea ca exemplu pentru utilizarea listelor circulare. O variantă a acestei probleme este aşa numita problemă a lui *Josephus*. Ea se formulează ca mai jos.

O cetate este apărată de un număr de soldaţi care îşi dau seama că au nevoie de ajutoare pentru a rezista în faţa duşmanului care îi atacă. Se pune problema de a alege pe unul dintre ei care să plece după ajutor.

Alegerea se face aşezînd soldaţii în cerc şi trăgînd la sorţi numele soldatului de la care să înceapă numărătoarea. De asemenea, se trage la sorţi un număr întreg n > 1. Se numără, în sensul acelor ceasornicului, începînd cu soldatul următor celui al cărui nume a fost tras la sorţi şi al n-lea soldat este scos din cerc. Se continuă numărătoarea în acelaşi fel începînd cu soldatul care urmează după cel scos din cerc. În felul acesta, după un număr finit de paşi, cercul se reduce la un singur soldat căruia îi revine sarcina să plece după ajutoare.

Problema lui Josephus este o variantă a formulării descrise prin punctele 1-2 de mai sus, dacă se consideră că pointerul *cuv* pointează spre numele unui soldat. La punctul 1 se precizează că se porneşte cu un nod care urmează imediat nodului care conţine pointerul spre un şir dat. Acest şir dat, este chiar numele soldatului tras la sorţi, care apare în problema lui Josephus.

Programul de mai jos rezolvă această problemă conform următorilor paşi:

a. Citeşte numărul *n* indicat în formularea problemei.

b. Citeşte un şir de caractere care defineşte nodul de la care începe numărătoarea.

c. Crează lista circulară care trebuie să conţină un nod pentru care *cuv* pointează spre un şir identic cu cel citit la punctul b.

d. Se elimină nodurile listei circulare conform paşilor 1 şi 2 indicaţi mai sus.

e. Se afişează cuvîntul din nodul rămas în listă.

PROGRAMUL BXI14

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <alloc.h>

typedef struct tnod {
 char *cuv;
 struct tnod *urm;
} TNOD;

#include "bviii2.cpp" /* pcit_int     */
#include "bviii3.cpp" /* pcit_int_lim */
#include "bxi10.cpp"  /* incnod       */
#include "bxi11.cpp"  /* elibnod      */
#include "bxi12.cpp"  /* ccrelist     */
#include "bxi13.cpp"  /* ccncs        */

#define MAXN 1000
TNOD *ptrnod;

main() /* - creeaza o lista circulara si elimina nodurile ei pornind de la
            un nod dat si eliminind tot al n-lea nod pina cind lista se
            reduce la un singur nod;
          - in final se listeaza sirul spre care pointeaza cuv al nodului
            la care s-a redus lista. */
{
 char t[255];
 int i,n;
 char er[]="s-a tastat EOF\n";
 TNOD *p,*p1;
```

```
/* citeste pe n */
 if(pcit_int_lim("n= ",2,MAXN,&n) == 0 ) {
     printf(er);
     exit(1);
 }

/* citeste un sir de caractere care va defini nodul din lista pentru pornirea
   numararii nodurilor */
 printf("sirul pentru pornirea numararii\n");
 if(gets(t) == 0 ) {
     printf(er);
     exit(1);
 }

/* creaza lista circulara */
 printf("tastati sirurile care intra in\
        compunerea listei\n");
 printf("cite un sir pe un rind\n");
 printf("la sfirsit se tasteaza Ctrl-Z\n");
 ccrelist();

/* se determina nodul pentru care cuv pointeaza spre un sir identic cu cel
   pastrat in tabloul t */
 if((p = ccncs(t)) == 0 ) {

/* nu exista un nod pentru care sirul spre care pointeaza cuv sa coincida
   cu cel pastrat in t */
     printf("nu se poate determina nodul de la\
            care sa se inceapa numararea\n");
     exit(1);
 }

/* elimina nodurile din lista pina se ajunge la un singur nod */
 p=p ->urm; /* numaratoarea incepe cu nodul urmator celui pentru
               care cuv pointeaza spre un sir identic cu cel pastrat
               in t */
 while (ptrnod != ptrnod -> urm ) {

/* lista contine mai mult de un nod */

/* se cauta al n-lea nod incepind cu cel spre care pointeaza p */
     for( i=1; i<n; i++) {
         p1 = p;
```

```
        p = p -> urm;
    }
```

```
/* - se sterge nodul spre care pointeaza p;
   - p1 pointeaza spre nodul precedent nodului spre care pointeaza p. */
    p1 ->urm = p -> urm;
    if(ptrnod == p )
        ptrnod = p1;
    free( p -> cuv);
    free(p);
    p = p1 -> urm; /* numaratoarea continua incepind cu
                      nodul urmator celui sters */
  }

/* lista s-a redus la un singur nod */
 printf("sirul cautat: \n");
 printf("%s\n", ptrnod -> cuv );
}
```

11.4. Listă dublu înlănţuită

Atît listele simplu înlănţuite, cît şi listele circulare, conduc adesea la parcurgeri neefective ale lor. De exemplu, ştergerea ultimului nod al unei liste simplu înlănţuite implică parcurgerea listei respective (vezi funcţia *sun*).

Astfel de parcurgeri pot fi uneori evitate folosind liste dublu înlănţuite. Într-o astfel de listă fiecare nod conţine doi pointeri: unul spre nodul *următor* şi unul spre nodul *precedent*.

În paragraful de faţă, vom presupune că nodurile listelor dublu înlănţuite au tipul definit ca mai jos:

typedef struct *tnod* {
    declaraţii
    struct *tnod* **prec*;
    struct *tnod* **urm*;
}*TNOD*;

Pentru a gestiona o listă dublu înlănţuită vom utiliza variabilele globale *prim* şi *ultim*, ca în cazul listelor simplu înlănţuite. Variabila *prim* pointează spre nodul pentru care prim-> prec=0. Pentru restul nodurilor, pointerul *prec* al unui nod pointează spre nodul precedent al listei.

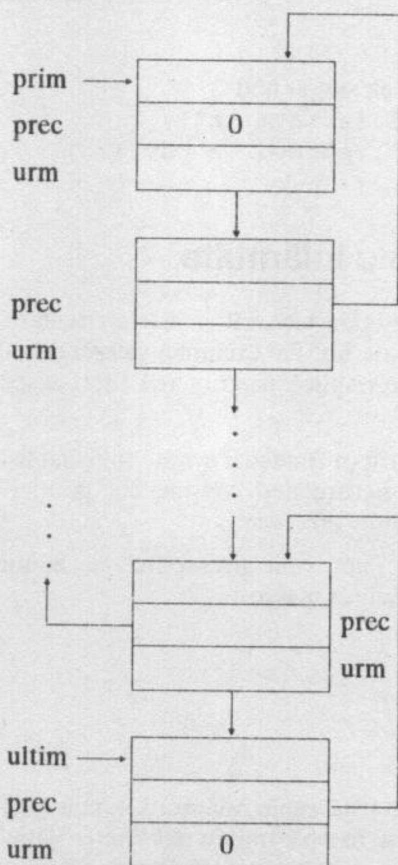
Variabila *ultim* pointează spre un nod pentru care ultim -> urm=0.

Pentru restul nodurilor, pointerul *urm* al unui nod pointează spre nodul următor al listei.

În concluzie, fiecare nod al listei are un nod precedent definit prin pointerul *prec* şi un următor definit prin pointerul *urm*.

O excepţie de la această regulă o constituie nodurile spre care pointează variabilele *prim* şi *ultim*. Nodul spre care pointează *prim* nu are precedent, iar nodul spre care pointează *ultim* nu are următor. Aceste noduri constituie capetele listei dublu înlănţuite.

În figura de mai jos se dă o schemă pentru listele dublu înlănţuite.



Listă dublu înlănţuită

În legătură cu listele dublu înlănţuite se pot defini aceleaşi operaţii ca şi în cazul listelor simplu înlănţuite:

a. crearea unei liste dublu înlănţuite;

b. accesul la un element al unei liste dublu înlănţuite;

c. inserarea unui nod într-o listă dublu înlănţuită;

d. ştergerea unui nod dintr-o listă dublu înlănţuită;

e. ştergerea unei liste dublu înlănţuite.

Operaţiile de la punctele *b* şi *e* se realizează ca în cazul listelor simplu înlănţuite şi de aceea ele nu vor mai fi considerate în paragrafele care urmează.

11.4.1. Crearea unei liste dublu înlănţuite

Definim mai jos funcţia *dcrelist* utilizată pentru a crea o listă dublu înlănţuită. Ea este analogă cu funcţia *crelist* definită în paragraful 11.1.1.

Funcţiile *incnod* şi *elibnod* au aceeaşi semnificaţie şi utilizare ca în cazul funcţiei *crelist*.

```
int dcrelist() /* - creaza o lista dublu inlantuita;
                  - returneaza:
                       0 - la eroare;
                      -1 - creare normala. */
{
 extern TNOD *prim,*ultim;
 int i,n;
 TNOD *p;

 n=sizeof(TNOD);
 prim=ultim=0;
 while(((p=(TNOD *)malloc(n))!=0)&&
       ((i=incnod(p))==1))
     if(prim==0){
        prim=ultim=p;
        p -> prec=p -> urm=0;
     }
     else{
        ultim -> urm=p;
        p -> prec=ultim;
        p -> urm=0;
        ultim=p;
```

```
    }
  if(p==0){
      printf("memorie insuficienta\n");
      exit(1);
  }
  elibnod(p);
  return i;
}
```

11.4.2. Inserarea unui nod într-o listă dublu înlănţuită

Într-o listă dublu înlănţuită se pot face inserări în diferite poziţii. În acest paragraf vom considera cîteva posibilităţi ca în cazul listelor simplu înlănţuite:

a. inserare înaintea primului nod al listei (nodul spre care pointează variabila *prim* este primul nod al listei);

b. inserare înaintea unui nod precizat printr-o cheie;

c. inserare după un nod precizat printr-o cheie;

d. inserare după ultimul nod al listei (nodul spre care pointează variabila *ultim* este ultimul nod al listei).

11.4.2.1. Inserarea unui nod într-o listă dublu înlănţuită înaintea primului nod al ei

```
TNOD *diniprim() /* - insereaza nodul curent inaintea primului
                      nod al listei;
                    - returneaza pointerul spre nodul inserat. */
{
 extern TNOD *prim,*ultim;
 TNOD *p;
 int n;

 n=sizeof(TNOD);
 if(((p=(TNOD *p)malloc(n))!=0)&&(incnod(p)==1)){
     if(prim==0){
         prim=ultim=p;
         p -> prec = p -> urm=0;
     }
     else{
         p -> urm = prim;
         p -> prec=0;
         prim -> prec = p;
```

```
            prim=p;
        }
        return p;
    }
    if(p==0){
        printf("memorie insuficienta\n");
        exit(1);
    }
    elibnod(p);
    return 0;
}
```

11.4.2.2. Inserarea unui nod într-o listă dublu înlănţuită înaintea unui nod precizat printr-o cheie

Vom presupune că, cheia este de tip *int*. Tipul TNOD se declară astfel:

typedef struct *tnod* {
    declaratii
    int *cheie*;
    declaratii
    struct *tnod* **prec*;
    struct *tnod* **urm*;
} *TNOD*;

Funcţia de inserare apelează funcţia *dcnci* care caută într-o listă dublu înlănţuită un nod de cheie dată. Această funcţie este identică cu funcţia *cnci* definită în paragraful 11.1.2.

```
TNOD *dcnci(int c)
/* - cauta un nod al listei pentru care cheie=c;
   - returneaza pointerul spre nodul determinat in acest fel sau zero daca
     nu exista nici un nod pentru care cheie=c. */
{
 extern TNOD *prim;
 TNOD *p;

 for(p=prim;p;p=p -> urm)
     if(p -> cheie==c)
         return p;
 return 0;
}
```

```
TNOD *dinici(int c)
/* - insereaza un nod inaintea unui nod precizat printr-o cheie numerica;
```

```
- returneaza pointerul spre nodul inserat sau zero daca nu are loc
  inserarea. */
{
 extern TNOD *prim;
 TNOD *p,*q;
 int n;

/* cauta nodul pentru care cheie=c */
 if((q=dcnci(c))==0){
    printf("nu exista in lista un nod de\
          cheie=%d\n",c);
    return 0;
 }
 n=sizeof(TNOD);
 if(((p=(TNOD *)malloc(n))!=0)&&(incnod(p)==1)){
    p -> prec=q -> prec;
    p -> urm=q;
    if(q -> prec!=0)

/* precedentul lui q are ca urmator nodul inserat */
        q -> prec -> urm=p;
    q -> prec=p;
    if(prim==q)

/* s-a inserat inaintea primului nod */
        prim=p;
    return p;
 }
 if(p==0){
    printf("memorie insuficienta\n");
    exit(1);
 }
 elibnod(p);
 return 0;
}
```

11.4.2.3. Inserarea unui nod într-o listă dublu înlănţuită după unul precizat printr-o cheie

Vom presupune că, cheia este de tip *int*, iar tipul TNOD este cel declarat în paragraful precedent. De asemenea, pentru a localiza nodul de cheie precizată, se va utiliza funcţia *dcnci* definită în acelaşi paragraf.

```
TNOD *dindci(int c)
/* - insereaza un nod dupa unul precizat printr-o cheie numerica;
   - returneaza pointerul spre nodul inserat sau zero daca inserarea nu
     are loc. */
{
 extern TNOD *ultim;
 TNOD *p,*q;
 int n;

 if((q=dcnci(c))==0){
     printf("nu exista nodul de cheie=%d\n",c);
     return 0;
 }
 n=sizeof(TNOD);
 if(((p=(TNOD *p)malloc(n))!=0)&&(incnod(p)==1)){
     p -> prec=q;
     p -> urm=q -> urm;
     if(q -> urm!=0)

/* urmatorul lui q are ca si precedent nodul inserat */
         q -> urm -> prec=p;
     q -> urm=p;
     if(ultim==q)

/* s-a inserat dupa ultimul nod */
         ultim=p;
     return p;
 }
 if(p==0){
     printf("memorie insuficienta\n");
     exit(1);
 }
 elibnod(p);
 return 0;
}
```

11.4.2.4. Inserarea unui nod într-o listă dublu înlănţuită după ultimul nod (adăugarea unui nod la o listă dublu înlănţuită)

În acest caz tipul nodurilor se declară astfel:

typedef struct *tnod* {
    declaratii
    struct *tnod* **prec*;

```
    struct tnod *urm;
} TNOD;
```

nefiind necesară prezenţa unei chei.

```
TNOD *dadauga()
/* - adauga un nod la o lista dublu inlantuita;
   - returneaza pointerul spre nodul inserat sau zero daca nu se
     realizeaza inserarea. */
{
 extern TNOD *prim,*ultim;
 TNOD *p;
 int n;

 n=sizeof(TNOD);
 if(((p=(TNOD *)malloc(n))!=0)&&(incnod(p)==1)){
     if(prim==0){
         prim=ultim=p;
         p -> prec=p -> urm=0;
     }
     else{
         ultim -> urm=p;
         p -> prec = ultim;
         p -> urm=0;
         ultim=p;
     }
     return p;
 }
 if(p==0){
     printf("memorie insuficienta\n");
     exit(1);
 }
 elibnod(p);
 return 0;
}
```

11.4.3. Ştergerea unui nod dintr-o listă dublu înlănţuită

Dintr-o listă dublu înlănţuită se pot şterge noduri. În cele ce urmează vom avea în vedere următoarele cazuri:

a. ştergerea primului nod al unei liste dublu înlănţuite;

b. ştergerea unui nod precizat printr-o cheie;

c. ştergerea ultimului nod al unei liste dublu înlănţuite.

11.4.3.1. Ştergerea primului nod al unei liste dublu înlănţuite

```
void dspn() /* sterge primul nod din lista */
{
 extern TNOD *prim,*ultim;
 TNOD *p;

 if(prim==0)
     return;
 p=prim;
 prim=prim -> urm;
 elibnod(p);
 if(prim==0)
     ultim=0; /* lista a devenit vida */
 else
     prim -> prec=0;
}
```

11.4.3.2. Ştergerea unui nod dintr-o listă dublu înlănţuită precizat printr-o cheie

Vom considera că, cheia este de tip *int*. În acest caz tipul TNOD se declară ca în paragraful 11.4.2.2.

Funcţia de faţă apelează funcţia *dcnci*, definită în acelaşi paragraf, pentru a localiza nodul care urmează a fi şters.

```
void dsnci(int c) /* sterge nodul de cheie=c */
{
 extern TNOD *prim,*ultim;
 TNOD *p;

 if(prim==0) /* lista vida */
    return;
 if((p=dcnci(c))==0){
    printf("lista nu contine nodul\
          de cheie = %d\n",c);
    return;
 }
 if(prim==p&&ultim==p){

/* lista are un singur nod; devine vida */
    prim=ultim=0;
    elibnod(p);
```

```
        return;
    }
    if(prim==p){ /* se sterge primul nod din lista */
        prim=prim -> urm;
        prim -> prec=0;
        elibnod(p);
        return;
    }
    if(ultim==p){ /* se sterge ultimul nod */
        ultim=ultim -> prec;
        ultim -> urm=0;
        elibnod(p);
        return;
    }

  /* se sterge un nod diferit de capete */

  /*urmatorul nodului care se sterge are ca precedent, precedentul nodului
    care se sterge */
   p -> urm -> prec=p -> prec;

  /* precedentul nodului care se sterge are ca urmator, urmatorul nodului
     care se sterge */
   p -> prec -> urm=p -> urm;
   elibnod(p);
  }
```

11.4.3.3. Ştergerea ultimului nod al unei liste dublu înlănţuite

În acest caz tipul TNOD nu necesită prezenţa unei chei. Se poate utiliza declaraţia indicată în paragraful 11.4.2.4.

```
  void dsun() /* sterge ultimul nod din lista */
  {
   extern TNOD *prim,*ultim;
   TNOD *p;

   if(prim==0)
      return;
   p=ultim;
   ultim=ultim -> prec;
   if(ultim==0)
      prim=0; /* lista devine vida */
   else
```

```
    ultim -> urm=0;
  elibnod(p);
}
```

Observaţii:

1. Funcţia *dsun*, spre deosebire de funcţia *sun* relativă la listele simplu înlănţuite, devine eficientă deoarece ea nu mai necesită parcurgerea nodurilor listei.

2. Funcţiile de inserare înainte şi după un nod precizat printr-o cheie numerică întreagă sînt mai simple în cazul listelor dublu înlănţuite în comparaţie cu analogele lor pentru liste simplu înlănţuite deoarece ele folosesc funcţia *dcnci* pentru localizarea nodului în raport cu care se face inserarea.

3. Funcţia de ştergere a unui nod precizat printr-o cheie numerică întreagă este mai simplă în cazul listelor dublu înlănţuite în comparaţie cu funcţia corespunzătoare pentru liste simplu înlănţuite deoarece ea foloseşte funcţia *dcnci* pentru localizarea nodului care se şterge.

4. O listă dublu înlănţuită devine o *listă circulară dublu înlănţuită* dacă se fac atribuirile:

   ultim -> urm=prim;

   şi

   prim -> prec=ultim.

Pentru a gestiona o astfel de listă nu mai sînt necesare variabilele *prim* şi *ultim*, lista ne mai avînd capete. În locul lor se utilizează un pointer spre un nod arbitrar al listei, ca în cazul listelor circulare simplu înlănţuite.

În legătură cu listele circulare dublu înlănţuite se ridică aceleaşi probleme ca şi în cazul listelor circulare simplu înlănţuite.

Propunem cititorului să construiască funcţii pentru a realiza operaţiile de bază asupra listelor circulare dublu înlănţuite:

- creare;
- acces la un nod de cheie dată;
- inserări de noduri;
- ştergeri de noduri;
- ştergerea listei.

Exerciţii:

11.15 Se consideră tipul utilizator:

```
typdef struct tnod {
    char *cuvant;
    int frecventa;
    struct tnod *prec;
    struct tnod *urm;
} TNOD
```

Să se scrie funcţia *dadauga*, care permite adăugarea la o listă dublu înlănţuită a unui nod de tipul TNOD definit ca mai sus.

Se observă că funcţia *dadauga* definită în paragraful 11.4.2.4, este dependentă numai de componentele *prec* şi *urm* din TNOD. De aceea, funcţia *dadauga* de mai jos este identică cu cea din paragraful 11.4.2.4.

FUNCŢIA BXI15

```
TNOD *dadauga()
/* - adauga un nod la o lista dublu inlantuita;
   - returneaza pointerul spre nodul inserat sau zero daca nu se
     realizeaza inserarea. */
{
 extern TNOD *prim,*ultim;
 TNOD *p;
 int n;

 n = sizeof(TNOD);
 if(((p = (TNOD *)malloc(n)) != 0 ) &&
        (incnod(p) == 1)) {
     if( prim == 0 ) {
         prim = ultim = p;
         p -> prec = p -> urm = 0;
     }
     else {
         ultim -> urm = p;
         p -> prec = ultim;
         p -> urm = 0;
         ultim = p;
     }
     return p;
 }
 if( p == 0 ) {
```

```
        printf("memorie insuficienta\n");
        exit(1);
    }
    elibnod(p);
    return 0;
}
```

11.16 Să se scrie o funcţie care şterge ultimul nod al unei liste dublu înlănţuite ale cărei noduri au tipul TNOD definit în exerciţiul precedent.

Această funcţie este definită în paragraful 11.4.3.3.

FUNCŢIA BXI16

```
void dsun() /* sterge ultimul nod din lista*/
{
 extern TNOD *prim,*ultim;
 TNOD *p;

 if(prim == 0 )
    return;
 p = ultim;
 ultim = ultim -> prec;
 if(ultim == 0 )
    prim = 0;
 else
    ultim -> urm = 0;
 elibnod(p);
}
```

11.17 Să se scrie un program care citeşte cuvintele dintr-un text şi afişează numărul de apariţii al fiecărui cuvînt din textul respectiv.

Această problemă a mai fost rezolvată în exerciţiile 10.50 şi 11.6. În exerciţiul 11.6 s-a creat o listă simplu înlănţuită ale cărei noduri conţin pointeri spre cuvintele diferite din text, precum şi frecvenţa de apariţie a acestora în text.

În cazul de faţă, se utilizează o listă dublu înlănţuită. Prin aceasta, eficienţa programului este mai mare deoarece funcţia *dsun*, care şterge ultimul nod dintr-o listă dublu înlănţuită, este mult mai eficientă decît funcţia *sun* care realizează acelaşi lucru relativ la o listă simplu înlănţuită.

Programul de faţă utilizează o parte din funcţiile apelate de programul BXI6.CPP şi anume:

- citcuv;
- incnod;
- elibnod;

şi

- cncs.

Funcţiile *adauga* şi *sun* se înlocuiesc cu funcţiile *dadauga* şi respectiv *dsun*. De asemenea, TNOD se declară ca în exerciţiul 11.15.

PROGRAMUL BXI17

```
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
#include <string.h>

typedef struct tnod {
char *cuvant;
int frecventa;
struct tnod *prec;
struct tnod *urm;
} TNOD;

#include "bx48.cpp"   /* citcuv   */
#include "bxi1.cpp"   /* incnod   */
#include "bxi2.cpp"   /* elibnod  */
#include "bxi15.cpp"  /* dadauga */
#include "bxi4.cpp"   /* cncs     */
#include "bxi16.cpp"  /* dsun     */

TNOD *prim,*ultim;

main() /* citeste un text si afiseaza frecventa cuvintelor din textul
          respectiv */
{
 TNOD *p,*q;

 prim = ultim = 0;
 while((p = dadauga() ) != 0)
     if((q = cncs( p -> cuvant )) != ultim ) {
         q -> frecventa++;
         dsun();
```

```
   }
   for( p = prim; p; p = p -> urm)
      printf("cuvintul: %-51s are frecventa: %d\n",
   p -> cuvant, p -> frecventa);
}
```

11.18 Să se scrie un program care citeşte cuvintele dintr-un text şi scrie numărul de apariţie al fiecărui cuvînt, în ordinea alfabetică a cuvintelor respective.

Această problemă a fost rezolvată în exerciţiul 11.7. În exerciţiul respectiv s-a definit funcţia *ordlist* care s-a apelat înainte de a afişa frecvenţa cuvintelor citite. Ea a modificat înlănţuirile nodurilor listei simplu înlănţuite create prin citirea cuvintelor textului, în aşa fel încît cuvintele corespunzătoare nodurilor listei să fie ordonate alfabetic.

Programul definit în exerciţiul 11.7. diferă de cel defint în exerciţiul 11.6. prin prezenţa funcţiei *ordlist* şi apelul ei înainte de listarea rezultatului.

În exerciţiul de faţă se defineşte funcţia *dordlist* care realizează acelaşi lucru ca şi funcţia *ordlist*, adică modifică înlănţuirile nodurilor unei liste dublu înlănţuite.

Amintim că funcţia *ordlist* parcurge lista analizînd ordinea cuvintelor corespunzătoare nodurilor vecine. Dacă două cuvinte, care corespund la noduri vecine, nu sînt în ordine alfabetică, atunci se schimbă înlănţuirile nodurilor respective aşa încît ele să fie înlănţuite invers în listă.

Inversarea înlănţuirilor se realizează ca mai jos.

Presupunem că *p* pointează spre nodul curent din listă şi q=p -> urm, deci *q* pointează spre nodul următor din listă.

Dacă p -> cuvant pointează spre un cuvînt care este în ordine alfabetică după cuvîntul spre care pointează q -> cuvant, atunci nodurile *p* şi *q* se înlănţuiesc invers, deci *p* devine următorul lui q. Aceasta implică paşii:

1. Dacă p -> prec!=0, atunci p -> prec ->urm=q, deoarece următorul precedentului lui *p* devine *q*.
2. Dacă q -> urm!=0, atunci q -> urm ->prec=p, deoarece precedentul următorului lui *q* devine *p*.
3. p->urm=q -> urm, deoarece următorul lui *q* devine următorul lui *p*.
4. q -> urm=p, deoarece *p* devine următorul lui *q*.
5. q -> prec=p -> prec, deoarece precedentul lui *p* devine precedentul lui *q*.

6. p -> prec=q, deoarece *q* devine precedentul lui *p*.

PROGRAMUL BXI18

```
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
#include <string.h>

typedef struct tnod {
 char *cuvant;
 int frecventa;
 struct tnod *prec;
 struct tnod *urm;
} TNOD;

#include "bx48.cpp"   /* citcuv  */
#include "bxi1.cpp"   /* incnod  */
#include "bxi2.cpp"   /* elibnod */
#include "bxi15.cpp"  /* dadauga */
#include "bxi4.cpp"   /* cncs    */
#include "bxi16.cpp"  /* dsun    */

void dordlist()
/* ordoneaza nodurile listei de tip TNOD in asa fel incit cuvintele
   corespunzatoare nodurilor sa fie in ordine alfabetica */
{
 extern TNOD *prim,*ultim;
 TNOD *p,*q;
 int ind;
 if(prim == 0 )
     return ;
 ind = 1;
 while(ind) {
     ind = 0;
     for(p=prim; p -> urm; ) {
         q = p -> urm; /* q este urmatorul lui p */
         if(stricmp(p -> cuvant, q -> cuvant ) > 0){

/* se inverseaza inlantuirile nodurilor p si q */
             if(p->prec)
                 p->prec->urm=q;
             if(q->urm)
```

```
                q->urm->prec=p;
            p -> urm = q -> urm;
            q -> urm = p;
            q -> prec = p -> prec;
            p -> prec = q;
            if(p == prim)
                prim = q;
            if(q == ultim)
                ultim = p;
            ind = 1;
        }
        else /* se trece la nodul urmator deoarece cuvintele sint in
                ordine alfabetica */
            p = q;
    } /* sfirsit for */
  } /* sfirsit while */
} /* sfirsit dordlist */

TNOD *prim,*ultim;

main() /* citeste un text si afiseaza frecventa cuvintelor diferite in
          ordine alfabetica */
{
 TNOD *p,*q;

 prim = ultim = 0;
 while((p = dadauga() ) != 0)
     if((q = cncs(p-> cuvant)) != ultim ) {
         q -> frecventa++;
         dsun();
     }
 dordlist();
 for( p = prim; p; p=p -> urm)
     printf("cuvintul: %-51s are frecventa: %d\n",
            p -> cuvant, p -> frecventa);
}
```

12. ARBORI

Arborii, ca şi listele, sînt structuri de date de natură recursivă şi dinamică. În multe publicaţii de specialitate (vezi de exemplu [3]) arborele se defineşte recursiv.

Prin *arbore* înţelegem o mulţime *finită* şi *nevidă* de elemente numite noduri:

A = {A1,A2,...,An}, n > 0

care au următoarele proprietăţi:

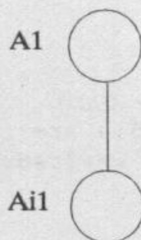
- Există un nod şi numai unul care se numeşte *rădăcina* arborelui.
- Celelalte noduri formează submulţimi disjuncte ale lui A, care formează fiecare cîte un arbore. Arborii respectivi se numesc *subarbori* ai rădăcinii.

Un arbore poate fi reprezentat într-un plan. Nodurile se reprezintă prin cercuri. Aşa de exemplu, dacă A1 este rădăcina arborelui A, atunci figurăm în plan un cerc pe care îl marcăm cu A1.

Fie acum:

{Ai1,Ai2,...,Aik}

un subarbore a lui A1. Deoarece acesta este el însuşi un arbore, fie Ai1 rădăcina lui. Lui Ai1 îi corespunde un cerc pe care îl marcăm cu Ai1. Acest cerc se figurează sub cercul corespunzător lui A1. Cele două cercuri se unesc ca în figura de mai jos.

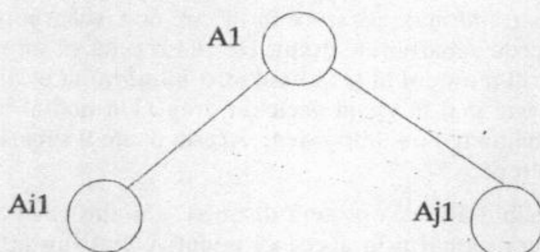


În mod analog, dacă:

{Aj1,Aj2,...,Ajr}

este un alt subarbore al rădăcinii A1 şi Aj1 este rădăcina acestui subarbore,

atunci lui Aj1 îi corespunde un cerc care se trasează sub cercul corespunzător lui A1. De asemenea, cercul marcat cu A1 se uneşte şi cu cel marcat cu Aj1. De obicei, cercurile marcate cu Ai1 şi Aj1 se află pe aceeaşi orizontală, ca în figura de mai jos.



Acest proces continuă cu toţi subarborii rădăcinii A1. Apoi se continuă cu subarborii lui Ai1, Aj1 şi aşa mai departe.

Într-un arbore există noduri cărora nu le mai corespund subarbori. Un astfel de nod se numeşte *terminal* sau *frunză*.

În legătură cu arborii s-a încetăţenit un limbaj conform căruia un nod *rădăcină* se spune că este un nod *tată*, iar subarborii rădăcinii sînt descendenţii acestuia. Rădăcinile descendenţilor unui nod tată sînt *fiii* lui. Astfel, în exemplul de mai sus, A1 este un nod tată, iar Ai1 şi Aj1 sînt fii ai acestuia.

Deci, dacă A este un nod rădăcină şi dacă acesta are *p* subarbori a căror rădăcini sînt:

B1,B2,...,Bp

atunci A este un nod *tată*, iar B1,B2,...,Bp sînt *fiii* lui.

Despre nodurile B1,B2,...,Bp se spune că sînt *fraţi*.

Numărul fiilor unui nod tată este *ordinul* nodului tată respectiv.

În exemplul de mai sus, nodul A are ordinul *p*. Un nod frunză nu are fii, deci ordinul lui este egal cu zero.

O altă noţiune legată de arbori este noţiunea de *nivel*. Rădăcina unui arbore are nivelul 1. Dacă un nod are nivelul *n*, atunci fiii lui au nivelul n+1.

Dacă pentru fiecare nod, subarborii săi sînt ordonaţi (se consideră într-o anumită ordine), atunci arborele se numeşte *ordonat*. În acest caz se obişnuieşte să se spună că rădăcina primului subarbore este *fiul cel mai în vîrstă*, iar rădăcina ultimului subarbore este *fiul cel mai tînăr*.

În multe aplicații practice întîlnim așa numiții *arbori binari*. Un arbore binar este o mulțime finită de elemente care sau este *vidă* sau conține un element numit *rădăcină*, iar celelalte elemente se împart în două submulțimi disjuncte, care fiecare la rîndul ei, este un arbore binar. Una din submulțimi este numită *subarborele stîng* al rădăcinii, iar cealaltă *subarborele drept*. Arborele binar este ordonat, deoarece în fiecare nod, subarborele stîng se consideră că precede subarborele drept. De aici rezultă că un nod al unui arbore binar are cel mult doi fii și că unul este *fiul stîng*, iar celălalt este *fiul drept*. Fiul stîng este mai în vîrstă decît cel drept. Un nod al unui arbore binar poate să aibă numai un descendent. Acesta poate fi subarborele stîng sau subarborele drept.

Cele două posibilități se consideră distincte. Cu alte cuvinte, dacă doi arbori binari diferă numai prin aceea că nodul A dintr-un arbore are ca descendent numai fiul stîng, iar același nod din celălalt arbore are ca descendent numai fiul drept, cei doi arbori se consideră distincți.

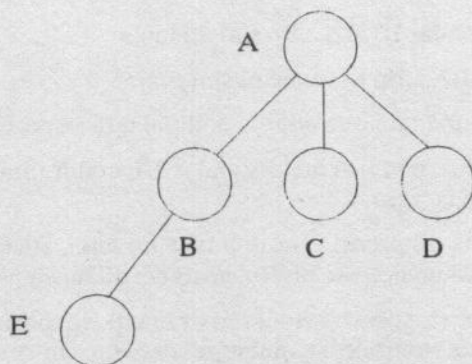
Un arbore binar *nu* se definește ca un caz particular de arbore ordonat. Astfel, un arbore nu este niciodată vid, spre deosebire de un arbore binar care poate fi și vid.

Un arbore ordonat poate fi totdeauna reprezentat printr-un arbore binar (vezi [3]).

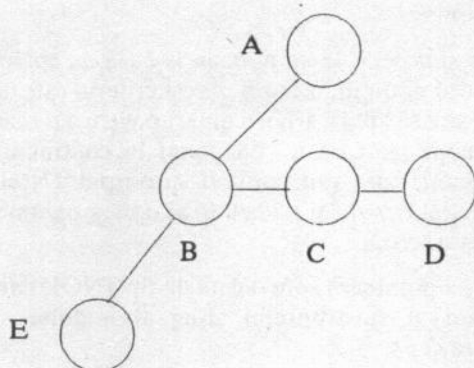
Transformarea se obține destul de simplu și anume:

> Se leagă împreună frații descendenți ai unui aceluiași nod tată și se suprimă legăturile lor cu nodul tată, exceptînd legătura primului dintre ei.

De exemplu, arborele din figura de mai jos:



se transformă în arborele binar:



În arborele transformat:

A are pe B ca fiu stîng;
B are pe C ca fiu drept şi pe E ca fiu stîng;
C are pe D ca fiu drept.

Prin regula de mai sus, rezultă că primul fiu al unui nod tată devine fiul stîng (cel mai vîrstnic). Celelalte legături formează subarbori drepţi: al doilea fiu devine fiul drept al primului, al treilea fiu devine fiul drept al celui de al doilea şi aşa mai departe.

În continuare ne vom ocupa numai de arborii binari.

Un nod al unui arbore binar este o dată structurată de tipul TNOD care se defineşte în felul următor:

```
typedef struct tnod {
      declaraţii
      struct tnod *st;
      struct tnod *dr;
} TNOD;
```

unde:

st - Este pointerul spre fiul stîng al nodului curent.

dr - Este pointerul spre fiul drept al aceluiaşi nod.

Asupra arborilor binari se pot defini mai multe operaţii dintre care amintim:

- inserarea unui nod frunză într-un arbore binar;
- accesul la un nod al unui arbore;
- parcurgerea unui arbore;

– ştergerea unui arbore.

Operaţiile de *inserare* şi *accesul* la un nod, au la bază un *criteriu* care să definească locul în arbore al nodului în cauză. Acest criteriu este dependent de problema concretă la care se aplică arborii binari pentru a fi rezolvată. El se defineşte printr-o funcţie pe care o vom numi în continuare funcţia *criteriu*. Ea are doi parametri care sînt pointeri spre tipul TNOD. Fie *p1* primul parametru al funcţiei *criteriu* şi *p2* cel de al doilea parametru al ei. Atunci, funcţia *criteriu* returnează:

-1 - Dacă *p2* pointează spre o dată de tip TNOD care *poate* fi un nod al subarborelui stîng al nodului spre care pointează *p1*.

1 - Dacă *p2* pointează spre o dată de tip TNOD care *poate* fi un nod al subarborelui drept al nodului spre care pointează *p1*.

0 - Dacă *p2* pointează spre o dată de tip TNOD care *nu* poate fi nod al subarborilor nodului spre care pointează *p1*.

Spre exemplificare, să considerăm că la intrare se află şirul de întregi:

20, 30, 5, 20, 4, 30, 7, 40.

şi se pune problema de a determina numărul de apariţii al fiecăruia.

Această problemă se poate rezolva dacă se construieşte un arbore binar ale cărui noduri au tipul TNOD definit astfel:

```
typedef struct tnod {
        int nr;
        int f;
        struct tnod *st;
        struct tnod *dr;
} TNOD
```

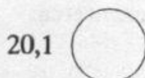
Variabila *nr* are ca valoare unul din numerele citite de la intrare, iar *f* reprezintă frecvenţa lui de apariţie în şirul de la intrare.

Într-un nod frunză st=dr=0 (pointerul nul).

La început se citeşte valoarea 20 şi arborele va conţine un singur nod, corespunzător valorii 20:

nr=20
f=1
st=dr=0

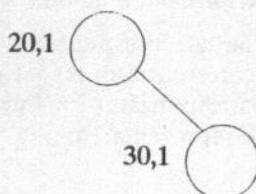
Figurăm arborele de la această fază astfel:



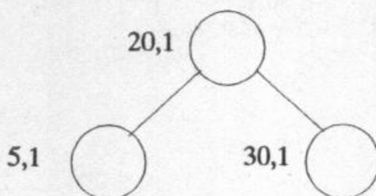
La pasul următor se citeşte întregul 30 şi se construieşte data de tip TNOD corespunzătoare lui:

nr=30
f=1
st=dr=0

Această dată o vom insera ca fiu drept al rădăcinii arborelui existent de la pasul precedent. Se obţine arborele:



Apoi se citeşte valoarea 5 şi nodul corespunzător acesteia se inserează ca fiu stîng al nodului corespunzător valorii 20. Se obţine arborele:

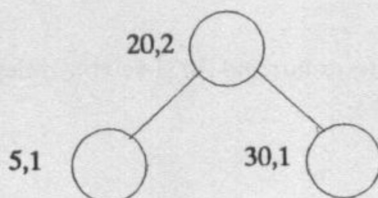


Nodurile se inserează în arbore în aşa fel încît dacă un nod corespunde întregului *n*, atunci subarborele stîng al lui conţine noduri care corespund la valori mai mici decît *n*, iar subarborele drept al aceluiaşi nod conţine noduri care corespund la valori mai mari decît *n*.

Se observă că arborele de mai sus respectă această regulă. Într-adevăr, rădăcina arborelui corespunde valorii 20, subarborele stîng al ei conţine, nodul corespunzător valorii 5 (5 < 20), iar subarborele drept al rădăcinii conţine nodul corespunzător valorii 30 (20 < 30).

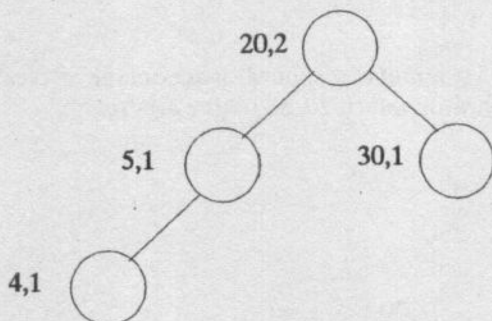
În continuare se citeşte valoarea 20 şi cum există deja în arbore un nod corespunzător acestei valori, se incrementează valoarea lui *f* din nodul

respectiv. În felul acesta se obţine reprezentarea:



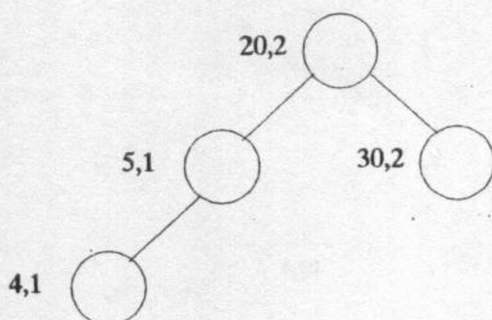
La pasul următor se citeşte valoarea 4. Se construieşte nodul corespunzător valorii 4. Deoarece 4 < 20, nodul curent trebuie inserat în subarborele stîng. Acesta este format din nodul corespunzător valorii 5.

Cum 4 < 5, rezultă că nodul corespunzător lui 4 trebuie inserat în subarborele stîng al nodului corespunzător lui 5. Cum nu există un astfel de subarbore, rezultă că nodul corespunzător lui 4 devine fiu stîng al nodului corespunzător lui 5. Se obţine reprezentarea:



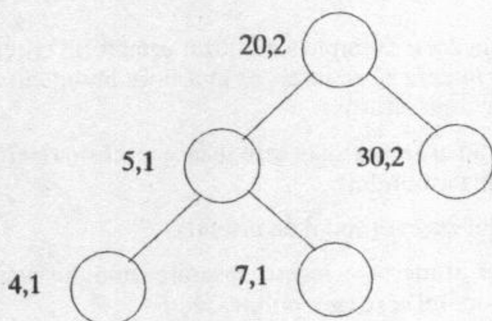
Valoarea următoare citită de la intrare este 30. Ca şi în cazurile precedente, se construieşte nodul corespunzător valorii citite 30.

Cum 30 > 20, rezultă că nodul construit curent trebuie să se insereze în subarborele drept al nodului corespunzător lui 20. Acest subarbore este format dintr-un singur nod care corespunde lui 30. Întrucît nodul curent corespunde aceleaşi valori, rezultă că el nu se mai inserează în arbore ci pur şi simplu se incrementează valoarea lui *f* pentru nodul corespunzător din arbore. Se obţine reprezentarea:



Următorul număr citit este 7. Se construieşte nodul corespunzător acestei valori. Deoarece 7 < 20, rezultă că nodul curent se va insera în subarborele stîng al nodului corespunzător lui 20. Rădăcina acestui subarbore este nodul corespunzător lui 5. Cum 5 < 7, rezultă că nodul curent se va insera în subarborele drept al nodului corespunzător lui 5. Întrucît nu există un astfel de subarbore, nodul curent se va insera ca fiu drept al nodului corespunzător lui 5.

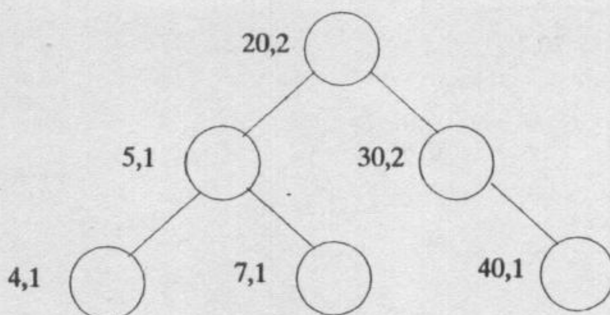
Arborele se reprezintă ca mai jos.



Ultima valoare citită este 40.

Raţionînd ca mai sus,se ajunge la concluzia că nodul corespunzător valorii 40 se inserează ca fiu drept al nodului corespunzător lui 30.

În final se obţine arborele de mai jos.



Se observă că arborele construit respectă regula enunţată mai sus.

Într-adevăr, nodul corespunzător lui 20 (n=20) are un subarbore stîng ale cărui noduri corespund valorilor: 5, 4 şi 7. Aceste valori sînt toate mai mici decît 20. Acelaşi nod, are un subarbore drept ale cărui noduri corespund valorilor: 30 şi 40. Ambele sînt mai mari decît 20.

Această proprietate are loc şi pentru celelalte noduri care nu sînt noduri frunză. De exemplu, nodul corespunzător valorii 5, are ca subarbore stîng nodul corespunzător lui 4 (4 < 5), iar ca subarbore drept nodul corespunzător lui 7 (5 < 7).

La construirea arborelui din acest exemplu s-a utilizat următorul criteriu pentru determinarea poziţiei în care să se insereze în arbore nodul curent (nodul corespunzător valorii curent citite):

a. p1 este pointer spre un nod al arborelui în care se face inserarea (iniţial p1 pointează spre rădăcina arborelui).

b. p2 este pointer spre nodul curent (nodul de inserat).

c. Dacă p2 -> nr < p1 -> nr, atunci se va încerca inserarea nodului curent în subarborele stîng al nodului spre care pointează *p1*.

d. Dacă p2 -> nr > p1 -> nr, atunci se va încerca inserarea nodului curent în subarborele drept al nodului spre care pointează *p1*.

e. Dacă p2 -> nr = p1 -> nr, atunci nodul curent nu se mai inserează în arbore deoarece există deja un nod corespunzător valorii curent citite.

   În acest caz se incrementează p1 -> f.

Acest criteriu se realizează imediat printr-o funcţie care are ca parametri pointerii *p1* şi *p2* şi care returnează valorile:

-1 - În cazul descris la punctul *c*.

1 - În cazul descris la punctul *d*.
0 - În cazul descris la punctul *e*.

Funcţia se defineşte ca mai jos:

```
int criteriu(TNOD *p1,TNOD *p2)
{
 if(p2 -> nr < p1 -> nr)
     return -1;
 if(p2 -> nr > p1 -> nr)
     return 1;
 return 0;
}
```

Din cele de mai sus rezultă că nodul curent nu se mai inserează în arbore, în situaţia descrisă de punctul *e*, situaţie care în general corespunde cazului cînd funcţia *criteriu* returnează valoarea zero. În acest caz nodurile spre care pointează *p1* şi *p2* le vom considera *echivalente*.

De obicei, la întîlnirea unei perechi de noduri echivalente, nodul din arbore (spre care pointează *p1*) este supus unei prelucrări, iar nodul curent (spre care pointează p2) este eliminat. Pentru a realiza o astfel de prelucrare este necesar să se apeleze o funcţie care are ca parametri pointerii *p1* şi *p2* şi care returnează un pointer spre tipul TNOD (de obicei se returnează valoarea lui *p1*).

Vom numi această funcţie *echivalenta*. Ea este dependentă de problema concretă, ca şi funcţia *criteriu*.

Pentru exemplul de mai sus, funcţia *echivalenta* se defineşte astfel:

```
TNOD *echivalenta(TNOD *p1,TNOD *p2)
{
 p1 -> f++;
 elibnod(p2);
 return p1;
}
```

Funcţia *elibnod* este o altă funcţie dependentă de problema concretă şi care eliberează zonele de memorie ocupate de nodul spre care pointează parametrul ei. Această funcţie a fost întîlnită în capitolul precedent şi s-a utilizat în acelaşi scop pentru nodurile din compunerea listelor.

În sfîrşit, o altă funcţie dependentă de problema concretă este funcţia *incnod*, utilizată şi ea în funcţiile din capitolul precedent.

Ea se apelează pentru a încărca datele în nodul curent care urmează a fi inserat în arbore sau pentru a fi prelucrat de funcţia *echivalenta* dacă nu are

loc inserarea.

În continuare se definesc funcţii pentru operaţiile asupra arborilor, amintite mai sus. Toate funcţiile utilizează o variabilă globală care este un pointer spre rădăcina arborelui. Numim *prad* această variabilă. Ea se defineşte astfel:

```
TNOD *prad;
```

În cazul în care într-un program se prelucrează simultan mai mulţi arbori, nu se va mai utiliza variabila globală *prad*, interfaţa dintre funcţii realizîndu-se cu ajutorul unui parametru care este pointer spre tipul TNOD şi căruia i se atribuie, la apel, adresa nodului rădăcină al arborelui prelucrat prin funcţia apelată.

În paragrafele următoare se definesc funcţii care utilizează variabila globală *prad*.

Menţionăm că funcţiile *incnod*, *criteriu*, *echivalenta* şi *elibnod* pot să aibă şi alte denumiri, făcînd modificări corespunzătoare în funcţiile care le apelează. De asemenea, aceste funcţii pot fi utilizate prin intermediul parametrilor.

Propunem cititorului să realizeze funcţiile din paragrafele acestui capitol în aşa fel încît funcţiile *incnod*, *criteriu*, *echivalenta* şi *elibnod* să fie apelate prin intermediul parametrilor de tip pointer spre funcţie.

12.1. Inserarea unui nod frunză într-un arbore binar

Arborele în care se inserează nodul este definit de variabila globală *prad* care are ca valoare adresa de început a zonei de memorie în care se păstrează rădăcina arborelui.

În cazul în care arborele este vid, *prad* are valoarea zero.

Inserarea nodului se realizează conform următorilor paşi:

1. Se alocă zonă de memorie pentru nodul care urmează să se insereze în arbore.

   Fie *p* pointerul care are ca valoare adresa de început a zonei respective.

2. Se apelează funcţia *incnod*, cu parametrul *p*, pentru a încărca datele curente în zona spre care pointează *p*.

   Dacă *incnod* returnează valoarea 1, se trece la pasul 3.

   Altfel se revine din funcţie cu valoarea zero.

3. Se fac atribuirile:

   p -> st=p -> dr=0

   deoarece nodul de inserat este nod frunză.

4. q=prad.

5. Se determină poziţia, în arbore, în care să se facă inserarea.

   În acest scop se caută nodul care poate fi nod tată pentru nodul curent.

   i=criteriu(q,p).

6. Dacă i<0, se trece la pasul 7.

   Altfel se trece la pasul 8.

7. Se încearcă inserarea nodului spre care pointează *p* (nodul curent) în subarborele stîng al nodului spre care pointează *q*.

   Dacă q -> st are valoarea zero, atunci nodul spre care pointează *q* nu are subarbore stîng şi nodul curent devine fiu stîng al celui spre care pointează *q* (q -> st=p).

   Se revine din funcţie returnîndu-se valoarea lui *p*.

   Altfel se face atribuirea q=q -> st (se trece la fiul stîng al nodului spre care pointează *q*) şi se trece la pasul 5.

8. Dacă i > 0, se trece la pasul 9.

   Altfel se trece la pasul 10.

9. Se încearcă inserarea nodului spre care pointează *p* (nodul curent) în subarborele drept al nodului spre care pointează *q*.

   Dacă q-> dr are valoarea zero, atunci nodul spre care pointează *q* nu are subarbore drept şi nodul curent devine fiu drept al celui spre care pointează *q* (q ->dr=p).

   Se revine din funcţie returnîndu-se valoarea lui *p*.

   Altfel se face atribuirea q=q -> dr (se trece la fiul drept al nodului spre care pointează *q*) şi se trece la pasul 5.

10. Nodul curent nu poate fi inserat în arbore.

   Se apelează funcţia *echivalenta* şi se revine din funcţie cu valoarea returnată de funcţia *echivalenta*.

Definim mai jos funcţia care realizează paşii 1-10.

```
TNOD *insnod()
/* - insereaza un nod in arborele binar spre a carui radacina pointea-
     za prad;
   - returneaza pointerul spre nodul inserat, pointerul returnat de functia
     echivalenta sau zero daca nu sînt date de incarcat in nod sau la e-
     roare. */
{
 extern TNOD *prad;
 int i;
 int n;
 TNOD *p,*q;

 n=sizeof(TNOD);
 if(((p=(TNOD *)malloc(n))!=0) &&
       (incnod(p)==1)){ /* 1 */
     p -> st=p -> dr=0;
     if(prad==0){ /* arbore vid */
         prad=p; /* nodul curent devine radacina arborelui */
         return p;
     }

/* se determina pozitia in arbore a nodului si se face inserarea daca este
   cazul */
     q=prad;
     for(;;){
         if((i=criteriu(q,p))<0)
             if(q -> st==0){ /* se insereaza ca fiu sting */
                 q -> st=p;
                 return p;
             }
             else{ /* se continua cautarea pozitiei in subarborele
                         sting */
                 q=q -> st;
                 continue;
             }
         if(i>0)
             if(q -> dr==0){ /* se insereaza ca fiu drept */
                 q -> dr =p;
                 return p;
             }
             else{ /* se continua cautarea pozitiei in
                         subarborele drept */
```

```
            q=q -> dr;
            continue;
        }
```

/* - nu se face inserarea deoarece nu exista un nod in arbore pentru care nodul curent sa poata fi nod fiu (sting sau drept);
- se apelează funcţia echivalenta si se revine din functie. */

```
        return echivalenta (q,p);
    } /* sfirsit for */
 } /* sfirsit if 1 */

 if(p==0){
     printf("memorie insuficienta\n");
     exit(1);
 }
```

/* eroare la incarcarea datelor in nod sau nu mai sînt date */

```
 elibnod(p);
 return 0;

} /* sfirsit insnod */
```

12.2. Accesul la un nod al unui arbore binar

Accesul la un nod al unui arbore binar presupune existenţa unui criteriu care să permită localizarea în arbore a nodului respectiv. De obicei, se poate utiliza funcţia *criteriu* folosită la crearea arborelui. Cu ajutorul ei se poate realiza accesul la un nod din arbore echivalent cu un nod dat.

```
TNOD *cauta(TNOD *p)
/* - cauta in arborele spre a carui radacina pointeaza prad, nodul
     echivalent cu cel spre care pointeaza p;
   - returneaza pointerul spre nodul determinat sau zero daca nu exista
     un astfel de nod */
{
 extern TNOD *prad;
 TNOD *q;
 int i;

 if(prad==0)
     return 0; /* arbore vid */
 for(q=prad;q;)
```

```
        if((i=criteriu(q,p))==0)
            return q;
        else
            if(i<0)
                q=q -> st; /* se cauta in subarborele sting */
            else
                q=q -> dr; /* se cauta in subarborele drept */
    return 0; /* nu s-a gasit in arbore un nod echivalent cu cel spre
                 care pointeaza p */
}
```

12.3. Parcurgerea unui arbore binar

Prelucrarea informaţiei păstrată în nodurile unui arbore binar se realizează parcurgînd nodurile arborelui respectiv. Parcurgerea nodurilor unui arbore binar se poate face în mai multe moduri. Mai jos, indicăm trei moduri de parcurgere a arborilor binari:

- în preordine;
- în inordine;
- în postordine.

Parcurgerea în preordine înseamnă accesul la rădăcină şi apoi parcurgerea celor doi subarbori ai ei, întîi subarborele stîng, apoi cel drept. Subarborii, fiind ei înşişi arbori binari, se parcurg în acelaşi mod.

Parcurgerea în inordine înseamnă parcurgerea mai întîi a subarborelui stîng, apoi accesul la rădăcină şi în continuare parcurgerea subarborelui drept. Cei doi subarbori se parcurg în acelaşi mod.

Parcurgerea în postordine înseamnă parcurgerea mai întîi a subarborelui stîng, apoi a subarborelui drept şi în final accesul la rădăcina arborelui. Cei doi subarbori se parcurg în acelaşi mod.

Accesul la un nod permite prelucrarea informaţiei conţinute în nodul respectiv. În acest scop se poate apela o funcţie care este dependentă de problema concretă care se rezolvă cu ajutorul parcurgerii arborelui.

Numim *prelucrare* această funcţie. Ea este de prototip:

void *prelucrare*(*TNOD* **p*);

unde:

p - Este pointerul spre nodul a cărui informaţie se prelucrează.

Un exemplu simplu de funcţie *prelucrare* este funcţia care afişează informaţia conţinută în nodul spre care pointează *p*.

Reluînd exemplul de la începutul capitolului, funcţia *prelucrare* poate fi definită ca o funcţie care afişează componentele *nr* şi *f* ale nodului spre care pointează parametrul ei:

```
void prelucrare(TNOD *p)
{
 printf("numarul=%d aparitii=%d\n",p -> nr,p -> f);
}
```

Folosind această funcţie, să exemplificăm cele trei moduri de parcurgere a arborilor binari cu ajutorul arborelui construit la începutul capitolului. Conform celor spuse mai sus, accesul la un nod va însemna apelul funcţiei *prelucrare*, adică afişarea componentelor *nr* şi *f* ale nodului respectiv.

a. Parcurgerea în preordine

1. Se are acces la rădăcina arborelui, deci se afişează:

   numarul=20 aparitii=2

2. Se parcurge subarborele stîng în preordine.

   Aceasta conduce la următorii paşi:

2.1. Se are acces la rădăcină, adică se afişează:

   numarul=5 aparitii=1

2.2. Se parcurge subarborele stîng în preordine.

   Aceasta conduce la următorii paşi:

2.2.1. Se are acces la rădăcină, adică se afişează:

   numarul=4 aparitii=1

2.2.2. Se parcurge subarborele stîng.

   Acesta este vid de aceea se trece la pasul următor.

2.2.3. Se parcurge subarborele drept.

   Acesta este vid (nodul corespunzător lui 4 este frunză) şi de aceea se trece la pasul următor.

2.3. Se parcurge în preordine subarborele drept corespunzător lui 5.

   Aceasta conduce la următorii paşi:

2.3.1. Se are acces la rădăcină, adică se afişează:

numarul=7 aparitii=1

2.3.2. Se parcurge subarborele stîng.

Acesta este vid.

Se trece la pasul următor.

2.3.3. Se parcurge subarborele drept.

Acesta este vid.

Se trece la pasul următor.

3. Se parcurge în preordine subarborele drept corespunzător lui 20.

Aceasta conduce la următorii paşi:

3.1. Se are acces la rădăcină, adică se afişează:

numarul=30 aparitii=2

3.2. Se parcurge subarborele stîng.

Acesta este vid.

Se trece la pasul următor.

3.3. Se parcurge în preordine subarborele drept corespunzător lui 30.

Aceasta conduce la paşii:

3.3.1. Se are acces la rădăcină, adică se afişează:

numarul=40 aparitii=1

3.3.2. Se parcurge subarborele stîng.

Acesta este vid.

Se trece la pasul următor.

3.3.3. Se parcurge subarborele drept.

Acesta este vid.

Procesul de parcurgere al arborelui se încheie.

Se observă că numerele citite au fost afişate în ordinea:

20, 5, 4, 7, 30, 40.

Introducem următoarele notaţii:

Rnr - Arborele care are ca rădăcină nodul ce corespunde întregului *nr*.

SRnr - Subarborele stîng al arborelui *Rnr*.

DRnr - Subarborele drept al arborelui *Rnr*.

b. Parcurgere în inordine

1. Se parcurge subarborele SR20 în inordine.
Aceasta conduce la următorii paşi:

1.1. Se parcurge subarborele SR5 în inordine.
Aceasta conduce la următorii paşi:

1.1.1. Se parcurge subarborele SR4.
Acesta este vid.
Se trece la pasul următor.

1.1.2. Se are acces la R4, adică se afişează:
numarul=4 aparitii=1

1.1.3. Se parcurge subarborele DR4.
Acesta este vid.
Se trece la pasul următor.

1.2. Se are acces la R5, adică se afişează:
numarul=5 aparitii=1

1.3. Se parcurge subarborele DR5 în inordine.
Aceasta conduce la următorii paşi:

1.3.1. Se parcurge subarborele SR7.
Acesta este vid.
Se trece la pasul următor.

1.3.2. Se are acces la R7, adică se afişează:
numarul=7 aparitii=1

1.3.3. Se parcurge subarborele DR7.
Acesta este vid.
Se trece la pasul următor.

2. Se are acces la R20, adică se afişează:
numarul=20 aparitii=2

3. Se parcurge subarborele DR20 în inordine.

Aceasta conduce la următorii paşi:

3.1. se parcurge subarborele SR30.

Acesta este vid.

Se trece la pasul următor;

3.2. Se are acces la R30, adică se afişează:

numarul=30 aparitii=2

3.3. Se parcurge subarborele DR30 în inordine.

Aceasta conduce la următorii paşi:

3.3.1. Se parcurge subarborele SR40.

Acesta este vid.

Se trece la pasul următor.

3.3.2. Se are acces la R40, adică se afişează:

numarul=40 aparitii=1

3.3.3. Se parcurge subarborele DR40.

Acesta este vid.

Procesul de parcurgere al arborelui se încheie.

Numerele citite au fost afişate în ordine crescătoare:

4, 5, 7, 20, 30, 40

Acest lucru rezultă imediat din modul în care a fost construit arborele şi anume; SRnr se compune din noduri care corespund la numere mai mici decît *nr*, iar DRnr se compune din noduri care corespund la numere mai mari decît *nr*. Această proprietate are loc pentru orice nod. Conform definiţiei parcurgerii *inordine*, nodurile se afişează în ordinea:

- nodurile lui SRnr;
- nodul Rnr;
- nodurile lui DRnr.

Acest principiu se aplică în continuare la subarborii SRnr şi DRnr şi aşa mai departe. În felul acesta se obţine o metodă nouă de *sortare*. Ea este mai eficientă decît metoda bulelor în ceea ce priveşte numărul paşilor.

Sortarea se reduce la construirea arborelui binar şi apoi parcurgerea acestuia în inordine. Inversînd condiţiile de inserare a nodurilor în arbore, se va realiza o sortare în ordine descrescătoare a numerelor citite de la

intrare. Cu alte cuvinte, ordinea sortării se defineşte cu ajutorul funcţiei *criteriu*.

La ora actuală există o serie de algoritmi de sortare care sînt mai eficienţi decît acesta şi de aceea nu mai insistăm asupra lui (vezi[3]).

c. Parcurgere în postordine

1. Se parcurge SR20 în postordine.

   Aceasta conduce la următorii paşi:

1.1. Se parcurge SR5 în postordine.

   Aceasta conduce la următorii paşi:

1.1.1. Se parcurge SR4.

   Acesta este vid.

   Se trece la pasul următor.

1.1.2. Se parcurge DR4.

   Acesta este vid.

   Se trece la pasul următor.

1.1.3. Se are acces la R4, adică se afişează:

   numarul=4 aparitii=1

1.2. Se parcurge DR5 în postordine.

   Aceasta conduce la următorii paşi:

1.2.1. Se parcurge SR7.

   Acesta este vid.

1.2.2. Se parcurge DR7.

   Acesta este vid.

1.2.3. Se are acces la R7, adică se afişează:

   numarul=7 aparitii=1

1.3. Se are acces la R5, adică se afişează:

   numarul=5 aparitii=1

2. Se parcurge DR20 în postordine.

   Aceasta conduce la următorii paşi:

2.1. Se parcurge SR30.

Acesta este vid.

2.2. Se parcurge DR30 în postordine.

Aceasta conduce la următorii paşi:

2.2.1. Se parcurge SR40.

Acesta este vid.

2.2.2. Se parcurge DR40.

Acesta este vid.

2.2.3. Se are acces la R40, adică se afişează:

numarul=40 aparitii=1

2.3. Se are acces la R30, adică se afişează:

numarul=30 aparitii=2

3. Se are acces la R20, adică se afişează:

numarul=20 aparitii=2

Procesul de parcurgere a arborelui se încheie.

Numerele citite s-au afişat în ordinea:

4, 7, 5, 40, 30, 20

În următoarele 3 paragrafe se definesc funcţii, pentru parcurgerea arborilor în modurile indicate mai sus. Toate cele 3 moduri de parcurgere a arborilor binari au o natură recursivă. Într-adevăr, în oricare din cele 3 moduri se realizează următoarele activităţi, în fiecare nod al arborelui:

1. Prelucrează nodul N.
2. Parcurgerea în acelaşi mod a subarborelui stîng al nodului N.
3. Parcurgerea în acelaşi mod a subarborelui drept al nodului N.

Diferenţa dintre cele 3 moduri constă numai în ordinea în care se realizează activităţile 1-3 de mai sus:

1, 2, 3 - În preordine.
2, 1, 3 - În inordine.
2, 3, 1 - În postordine.

Natura recursivă a acestor moduri de parcurgere decurge din faptul că subarborii fiecărui nod se parcurg în acelaşi mod.

Avînd în vedere acest fapt, funcţiile de parcurgere a arborilor binari se

definesc cel mai simplu prin funcţii recursive.

Fiecare din aceste funcţii au ca parametru un pointer spre TNOD:

void fparc(TNOD *p);

Apelul unei astfel de funcţii, atribuie lui *p* adresa de început a zonei de memorie în care se păstrează rădăcina arborelui. Rezultă că funcţia *fparc* poate fi apelată printr-o instrucţiune de forma:

fparc(prad);

12.3.1. Parcurgerea arborilor binari în preordine

Numim *preord* funcţia care parcurge în preordine un arbore binar. Ea realizează următoarele:

- Dacă pointerul spre rădăcină nu este nul, atunci se execută paşii de mai jos:
  1. Se apelează funcţia *prelucrare* cu pointerul spre rădăcină.
  2. Fiul sting devine rădăcină şi se reaplează funcţia *preord* cu pointerul spre noua rădăcină.

     În felul acesta se va parcurge în preordine subarborele sting.
  3. Fiul drept devine rădăcină şi se reapelează funcţia *preord* cu pointerul spre noua rădăcină.

     În felul acesta se va parcurge în preordine subarborele drept.
- Dacă pointerul spre rădăcină este nul se revine din funcţie.

Aceşti paşi se transcriu imediat în limbajul C, ca mai jos:

```
void preord(TNOD *p) /* parcurge arborele binar in preordine */
{
 if(p!=0){
     prelucrare(p); /* prelucreaza radacina */
     preord(p -> st); /* parcurge subarborele sting in
                         preordine */
     preord(p -> dr); /* parcurge subarborele drept in
                         preordine */
 }
}
```

12.3.2. Parcurgerea arborilor binari în inordine

Numim *inord* funcţia care parcurge în inordine un arbore binar. Ea realizează aceeaşi paşi ca şi funcţia *preord*, dar în altă ordine. Dacă menţinem numerotarea paşilor definiţi în paragraful precedent, atunci funcţia *inord* realizează paşii respectivi în ordinea:

2, 1, 3

În felul acesta se obţine funcţia de mai jos:

```
void inord(TNOD *p) /* parcurge arborele binar in inordine */
{
 if(p!=0){
     inord(p -> st); /* parcurge subarborele sting in
                        inordine */
     prelucrare(p); /* prelucreaza radacina */
     inord(p -> dr); /* parcurge subarborele drept in
                        inordine */
 }
}
```

12.3.3. Parcurgerea arborilor binari în postordine

Numim *postord* funcţia care parcurge în postordine un arbore binar. Ea realizează aceeaşi paşi ca şi funcţiile *preord* şi *inord*, dar în altă ordine şi anume:

2, 3, 1

unde prin 1, 2, 3, am notat paşii definiţi în paragraful 12.3.

Se obţine funcţia de mai jos:

```
void postord(TNOD *p) /* parcurge arborele binar
                         in postordine */
{
 if(p!=0){
     postord(p -> st); /* parcurge subarborele sting in
                          postordine */
     postord(p -> dr); /* parcurge subarborele drept in
                          postordine */
     prelucrare(p); /* prelucreaza radacina */
 }
}
```

12.4. Ştergerea unui arbore binar

Pentru a şterge un arbore binar este necesară parcurgerea lui şi ştergerea fiecărui nod al arborelui respectiv.

Ştergerea unui nod se realizează apelînd funcţia *elibnod*. Arborele se parcurge în *postordine*. Rezultă că funcţia care şterge un arbore binar este asemănătoare cu funcţia *postord* definită în paragraful 12.3.3. Deosebirea constă în aceea că, pentru a prelucra rădăcina se apelează funcţia *elibnod* în locul funcţiei *prelucrare*.

```
void stergarb(TNOD *p) /* sterge arborele spre a carui radacina
                          pointeaza p */
{
 if(p!=0){
     stergarb(p -> st);
     stergarb(p -> dr);
     elibnod(p);
 }
}
```

Menţionăm că funcţia *stergarb* nu atribuie valoarea zero variabilei globale *prad*. Aceasta este necesar să se realizeze în funcţia care apelează funcţia *stergarb*.

Exerciţii:

12.1 Se consideră tipul TNOD definit ca mai jos:

```
typedef struct tnod {
    int nr;
    int f;
    struct tnod *st;
    struct tnod *dr;
} TNOD;
```

Să se scrie funcţia *incnod* care are antetul:

int incnod(TNOD *p);

şi care realizează următoarele:

- citeşte un întreg de tip *int* şi-l atribuie variabilei p -> nr;
- atribuie valoarea 1 variabilei p->f;
- returnează:

  -1 - La întîlnirea sfîrşitului de fişier.

  1 - Altfel.

Funcţia *incnod* apelează funcţia *pcit_int_lim* definită în exerciţiul 8.3.

FUNCŢIA BXII1

```
int incnod(TNOD *p) /* incarca nodul spre care pointeaza p */
{
 int n;

 if(pcit_int_lim ("nr= ", -32768, 32767,&n) == 0 )
    return -1; /* s-a intilnit EOF */
 p -> nr = n;
 p -> f = 1;
 return 1;
}
```

12.2 Să se scrie funcţia *elibnod* care eliberează zona de memorie ocupată de un nod de tipul TNOD definit în exerciţiul 12.1.

FUNCŢIA BXII2

```
void elibnod(TNOD *p) /* elibereaza zona de memorie ocupata
                         de nodul spre care pointeaza p */
{
 free(p);
}
```

12.3 Să se scrie funcţia *echivalenta* de prototip:

TNOD *echivalenta (TNOD *q,TNOD *p);

care realizează următoarele:

- eliberează zona de memorie ocupată de nodul spre care pointează p;
- incrementează valoarea variabilei q -> f;
  (TNOD este tipul definit în exerciţiul 12.1.);
- returnează valoarea pointerului q.

FUNCŢIA BXII3

```
TNOD *echivalenta (TNOD *q,TNOD *p)
/* - elibereaza zona de memorie ocupata de nodul spre care pointeaza p,
     incrementeaza valoarea variabilei q -> f;
```

```
   - returneaza valoarea lui q. */
{
 elibnod(p);
 q -> f++;
 return q;
}
```

12.4 Să se scrie funcţia *prelucrare* care afişează valorile componentelor *nr* şi *f* dintr-o dată de tip TNOD spre care pointează parametrul ei.

FUNCŢIA BXII4

```
void prelucrare(TNOD *p) /* afiseaza pe p -> nr si p -> f */
{
 printf("numarul= %d aparitii= %d\n",p -> nr,
        p -> f);
}
```

12.5 Să se scrie funcţia *criteriu* definită la începutul acestui capitol pentru datele de tipul TNOD.

FUNCŢIA BXII5

```
int criteriu(TNOD *p1,TNOD *p2)
/* returneaza:
     -1  - daca p2->nr < p1->nr;
      1  - daca p2->nr > p1->nr;
      0  - altfel. */
{
 if(p2 -> nr < p1 -> nr)

/* nodul spre care pointeaza p2 poate fi inserat in subarborele sting al
   nodului spre care pointeaza p1 daca subarborele respectiv nu conti-
   ne un nod echivalent cu cel spre care pointeaza p2 */
    return -1;

 if(p2 -> nr > p1 -> nr )

/* nodul spre care pointeaza p2 poate fi inserat in subarborele drept al
   nodului spre care pointeaza p1 daca subarborele respectiv nu conti-
   ne un nod echivalent cu cel spre care pointeaza p2 */
    return 1;
```

```
/* nodurile spre care pointeaza p1 si p2 sînt echivalente */
 return 0;
}
```

12.6 Să se scrie o funcţie care inserează un nod într-un arbore binar.

Nodurile arborelui au tipul TNOD.

Această funcţie nu depinde de structura nodurilor arborelui. Ea a fost definită în paragraful 12.1.

FUNCŢIA BXII6

```
TNOD *insnod()
/*- insereaza un nod in arborele binar spre a carui radacina pointeaza
    prad;
  - returneaza pointerul spre nodul inserat sau pointerul returnat
    de functia echivalenta daca nodul de inserat este echivalent cu unul
    deja aflat in arbore;
  - returneaza zero daca nu mai sînt date de incarcat sau la eroare. */
{
 extern TNOD *prad;
 int i;
 int n;
 TNOD *p,*q;

 n = sizeof(TNOD);
 if(((p=(TNOD *)malloc(n))!=0) &&
      (incnod(p)==1)) { /* 1 */
     p -> st = p -> dr = 0;
     if(prad == 0 ) { /* arbore vid */
         prad = p;
         return p;
     }
     q = prad;
     for ( ; ; ) {
         if((i = criteriu(q,p)) < 0)
             if(q -> st == 0 ) {
                 q -> st = p;
                 return p;
             }
             else {
                 q = q -> st;
                 continue;
```

```
            }
        if( i > 0 )
            if(q -> dr == 0 ) {
                q -> dr = p;
                return p;
            }
            else {
                q = q -> dr;
                continue;
            }
        return echivalenta(q,p);
    } /* sfirsit for */
  } /* sfirsit if 1 */
  if( p == 0 ) {
      printf("memorie insuficienta\n");
      exit(1);
  }
  elibnod (p);
  return 0;
}
```

12.7 Să se scrie o funcţie care parcurge un arbore binar în preordine.

Această funcţie nu depinde de tipul nodurilor arborelui binar şi ea a fost definită în paragraful 12.3.1.

FUNCŢIA BXII7

```
void preord(TNOD *p) /* parcurge arborele binar in preordine */
{
 if(p != 0 ) {
     prelucrare(p);
     preord( p -> st);
     preord( p -> dr);
 }
}
```

12.8 Să se scrie o funcţie care parcurge arborele binar în inordine.

Această funcţie nu depinde de structura nodurilor arborelui şi ea a fost definită în paragraful 12.3.2.

FUNCŢIA BXII8

```
void inord(TNOD *p) /* parcurge arborele binar in inordine */
{
 if( p != 0 ) {
     inord( p -> st);
     prelucrare(p);
     inord( p -> dr);
 }
}
```

12.9 Să se scrie o funcţie care parcurge un arbore binar în postordine.

Această funcţie nu depinde de structura nodurilor arborelui şi ea a fost definită în paragraful 12.3.3.

FUNCŢIA BXII9

```
void postord(TNOD *p)  /* parcurge arborele binar
                          in postordine */
{
 if( p != 0) {
     postord( p -> st);
     postord( p -> dr);
     prelucrare(p);
 }
}
```

12.10 Să se scrie un program care citeşte un şir de întregi de tip *int* şi afişează frecvenţa de apariţie a fiecărui număr citit.

Numerele sînt separate prin caractere albe, iar la sfîrşit se tastează sfîrşitul de fişier.

Rezolvăm această problemă cu ajutorul arborilor binari.

La început se citesc numerele de la intrare şi acestea se păstrează în nodurile unui arbore binar de tipul TNOD definit în exerciţiul 12.1.

Componenta *nr* are ca valoare numărul citit. La inserarea unui nod în arbore, f=1, deoarece nodul neexistînd în prealabil în arbore, înseamnă că nici numărul pe care-l conţine nodul respectiv n-a mai fost citit înainte.

Dacă la citirea unui număr se constată că acestuia îi corespunde deja un nod în arbore, atunci se incrementează valoarea lui *f* pentru nodul respectiv aflat în arbore.

După construirea arborelui binar, acesta se va parcurge în cele 3 moduri definite în paragraful 12.3., afişîndu-se frecvenţa întregilor citiţi.

Se va observa că parcurgerea în inordine afişează numerele citite în ordine crescătoare.

PROGRAMUL BXII10

```
#include <stdio.h>
#include <stdlib.h>

typedef struct tnod {
 int nr;
 int f;
 struct tnod *st;
 struct tnod *dr;
} TNOD;

#include "bviii2.cpp" /* pcit_int     */
#include "bviii3.cpp" /*pcit_int_lim  */
#include "bxii1.cpp"  /* incnod       */
#include "bxii2.cpp"  /* elibnod      */
#include "bxii3.cpp"  /* echivalenta  */
#include "bxii4.cpp"  /* prelucrare   */
#include "bxii5.cpp"  /* criteriu     */
#include "bxii6.cpp"  /* insnod       */
#include "bxii7.cpp"  /* preord       */
#include "bxii8.cpp"  /* inord        */
#include "bxii9.cpp"  /* postord      */

TNOD *prad;

main()
/* citeste un sir de numere si le pastreaza in nodurile unui arbore binar,
   apoi afiseaza frecventa de aparitie a fiecarui numar citit, parcurgind
   arborele in preordine, inordine si postordine */
{
/* construieste arborele binar */
 prad = 0;
 while( insnod() )
     ;

/* parcurge arborele in preordine */
 printf("\n\n\tpreordine\n\n");
```

```
 preord(prad);

/* parcurge arborele in inordine */
 printf("\n\n\tinordine\n\n");
 inord(prad);

/* parcurge arborele in postordine */
 printf("\n\n\tpostordine\n\n");
 postord(prad);
}
```

12.11 Se consideră tipul TNOD ca mai jos:

```
typedef struct tnod {
    char *cuvant;
    int frecventa;
    struct tnod *st;
    struct tnod *dr;
} TNOD
```

Să se scrie funcţia *echivalenta* pentru nodurile de tipul TNOD definit mai sus, care realizează următoarele:

Fie antetul:

TNOD *echivalenta(TNOD *q,TNOD *p)

- funcţia eliberează zona de memorie ocupată de nodul spre care pointează *p*;
- incrementează valoarea componentei: q -> frecventa;
- returnează valoarea lui q.

Această funcţie este similară cu funcţia definită în exerciţiul 12.3.

FUNCŢIA BXII11

```
TNOD *echivalenta(TNOD *q,TNOD *p)
/* - elibereaza zona de memorie ocupata de nodul spre care pointeaza p;
  - incrementeaza pe q -> frecventa;
  - returneaza valoarea lui q. */
{
 elibnod (p);
 q -> frecventa++;
 return q;
}
```

Observaţie:

Tipul TNOD definit mai sus este similar cu tipul TNOD definit în exerciţiul 11.1. Aceste două tipuri diferă numai prin componentele de înlănţuire.

În cazul de faţă se folosesc pointerii *st* şi *dr*, iar în exerciţiul 11.1 se utilizează pointerul *urm*.

Celelalte componente (*cuvant* şi *frecventa*) sînt identice pentru cele două tipuri de noduri. De aceea, funcţia *elibnod* definită în exerciţiul 11.2. poate fi utilizată şi pentru nodurile de tipul TNOD definit mai sus.

12.12 Să se scrie funcţia *prelucrare* care afişează datele care nu sînt de înlănţuire dintr-un nod de tipul TNOD definit în exerciţiul 12.11.

FUNCŢIA BXII12

```
void prelucrare(TNOD *p) /* afiseaza p -> cuvant si
                                p -> frecventa */
{
 static int n = 0;

 printf("cuvintul: %-51s are frecventa: %d\n",
       p -> cuvant, p -> frecventa);
 if(( n+1)%23 == 0 ) {
     printf("pentru a continua actionati o\
          tasta\n");
     getch();
 }
 n++;
}
```

12.13 Să se scrie funcţia *criteriu* de antet:

int criteriu(TNOD *p1,TNOD *p2)

şi care returnează valorile:

-1 - dacă p2 -> cuvant < p1 -> cuvant;

1 - dacă p2 -> cuvant > p1 -> cuvant;

0 - dacă p2 -> cuvant = p1 -> cuvant.

unde:

TNOD - Este tipul definit în exerciţiul 12.11.

FUNCŢIA BXII13

```
int criteriu(TNOD *p1, TNOD *p2)
/* returneaza:
    -1  - daca p2->cuvant < p1->cuvant;
     1  - daca p2->cuvant > p1->cuvant;
     0  - altfel. */
{
 int i;

 if((i = stricmp(p2 -> cuvant,p1 -> cuvant)) < 0 )
    return -1;
 else
    if( i > 0 )
       return 1;
    else
       return 0;
}
```

12.14 Să se scrie un program care citeşte cuvintele dintr-un text şi afişează numărul de apariţii al fiecărui cuvînt din textul respectiv.

Cuvîntul se defineşte ca o succesiune de litere mici şi/sau mari. Textul se termină prin sfîrşitul de fişier.

Această problemă a fost rezolvată în capitolul 11, folosind listele simplu înlănţuite şi dublu înlănţuite. Programul de faţă rezolvă această problemă folosind arborii binari.

Rezolvarea cu ajutorul arborilor este similară cu cea folosită în exerciţiul 12.10. Diferenţa constă în aceea că în cazul exerciţiului 12.10 se citesc numere de tip *int*, iar în cazul de faţă se citesc cuvinte.

Programul de faţă, ca şi cel din exerciţiul 12.10, apelează o serie de funcţii care sînt definite în diverse exerciţii. Astfel, funcţia *elibnod* este cea definită în exerciţiul 11.2. (vezi observaţia de la exerciţiul 12.11.). Funcţia *incnod*, citeşte un cuvînt şi-l păstrează în memoria *heap*, apoi atribuie pointerului *cuvant* adresa zonei de început a memoriei în care se păstrează cuvîntul respectiv, iar la componenta *frecventa* i se atribuie valoarea 1. Funcţia returnează valoarea -1 la întîlnirea sfîrşitului de fişier şi 1 în caz contrar. Această funcţie este definită în exerciţiul 11.1. Ea apelează funcţia *citcuv* definită în exerciţiul 10.48.

Celelalte funcţii apelate de program sînt definite în capitolul de faţă.

PROGRAMUL BXII14

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <string.h>

typedef struct tnod {
 char *cuvant;
 int frecventa;
 struct tnod *st;
 struct tnod *dr;
} TNOD;

#include "bx48.cpp"    /* citcuv      */
#include "bxi1.cpp"    /* incnod      */
#include "bxi2.cpp"    /* elibnod     */
#include "bxii11.cpp"  /* echivalenta */
#include "bxii12.cpp"  /* prelucrare  */
#include "bxii13.cpp"  /* criteriu    */
#include "bxii6.cpp"   /* insnod      */
#include "bxii8.cpp"   /* inord       */

TNOD *prad;

main()
/* citeste cuvintele dintr-un text si le afiseaza in ordine alfabetica
   impreuna cu frecventa de aparitie a lor in text */
{
/* construieste arborele binar */
 prad = 0;
 while(insnod() )
     ;

/* parcurge arborele in inordine */
 inord(prad);
}
```

Observaţie:

Problema de faţă se rezolvă mai eficient folosind arborii binari în locul listelor. Aceasta din cauză că operaţia de căutare a unui cuvînt într-o listă este ineficientă. Într-adevăr, căutarea unui cuvînt într-o listă simplu înlănţuită presupune parcurgerea secvenţială a nodurilor ei fie pînă la

întîlnirea nodului ce conţine cuvîntul căutat, fie pînă la sfîrşitul listei în cazul în care cuvîntul nu este conţinut nici într-un nod al listei.

Folosind arborii, procesul de căutare al unui cuvînt în arbore necesită, de obicei, mai puţini paşi. În fiecare nod se realizează o ramificare, deoarece se continuă căutarea fie în subarborele stîng, fie în cel drept, fie se constată că nodul respectiv conţine chiar cuvîntul căutat.

În felul acesta se observă că la căutarea unui cuvînt o parte din nodurile arborelui rămîn neparcurse.

Utilizarea arborilor devine ineficientă dacă la citirea cuvintelor se obţine un arbore *degenerat*. Un arbore binar este degenerat, dacă subarborii lui sînt toţi de acelaşi fel, de exemplu,toţi sînt subarbori stîngi sau toţi sînt subarbori drepţi. În acest caz, arborele binar este echivalent cu o listă simplu înlănţuită.

În problema de faţă se obţin arbori degeneraţi dacă, de exemplu, se citesc cuvintele în ordine alfabetică. Aceasta este puţin probabil cînd se citesc cuvinte dintr-un text normal.

13. TABELE

Noţiunea de *tabel* o definim ca o *colecţie* de elemente identificabile prin *chei* [3].

Elementele colecţiei sînt date de *acelaşi tip*, care în mod frecvent este un tip utilizator.

Prin *cheie* înţelegem, ca şi în cazul listelor, o componentă a elementelor tabelei care are valori *diferite* pentru elemente diferite. În felul acesta, un element din tabel se identifică exact prin valoarea cheii sale.

Elementele unei tabele se mai numesc şi *înregistrări*.

Ca exemple de tabele întîlnite frecvent în diferite aplicaţii amintim: tabele de cuvinte rezervate, tabele de simboluri, tabele de indecşi pentru fişiere etc.

O problemă importantă care se pune în legătură cu tabelele este aceea a *căutării* unei înregistrări de cheie dată.

Tabelele de cuvinte rezervate sînt exemple de tabele *fixe* cu un număr de înregistrări cunoscut dinainte.

Astfel de tabele se întîlnesc în compilatoare şi se utilizează pentru a determina dacă un identificator este sau nu un cuvînt rezervat. În acest caz, identificatorul citit din textul de intrare se caută în tabela de cuvinte rezervate pentru a stabili dacă acesta este un cuvînt rezervat sau definit de utilizator. De obicei, procedura de căutare returnează un cod dacă identificatorul căutat s-a aflat în tabela de cuvinte rezervate sau o valoare distinctă de orice cod (de exemplu -1) în cazul în care identificatorul respectiv nu este un cuvînt rezervat.

Tabelele de acest tip se recomandă a fi *ordonate*, deoarece în felul acesta putem aplica metode de căutare eficiente, ca de exemplu *căutarea binară*.

Tabelele de simboluri sînt exemple de tabele *dinamice*, care se construiesc pe măsură ce se constată că înregistrarea curentă nu se află în tabelă. Astfel de tabele necesită alte metode de căutare şi care în mod frecvent sînt urmate de inserarea în tabelă a înregistrării inexistente.

O tabelă dinamică poate fi organizată ca o listă simplu sau dublu înlănţuită, dar în acest caz eficienţa procedurii de căutare este foarte mică.

O altă posibilitate este de a organiza tabela inserînd fiecare înregistrare în nodul unui arbore binar. În acest caz, este nevoie de un *criteriu* pe care să-l satisfacă cheile înregistrărilor tabelei respective. Acest criteriu se utilizează

la localizarea în arbore a nodului corespunzător înregistrării curente.

În cazul în care nu există un nod corespunzător înregistrării respective, se inserează în arbore un astfel de nod. În felul acesta, în general, se va mări mult eficienţa procedurilor de căutare şi inserare a înregistrărilor în arbore. Criteriul care se află la baza căutării într-un arbore binar, de obicei,reduce substanţial numărul nodurilor consultate. Astfel, în fiecare nod are loc una din următoarele 4 posibilităţi:

a. Nodul corespunde înregistrării curente.

   Procesul de căutare se încheie, identificîndu-se nodul căutat.

b. Căutarea se continuă în subarborele stîng al nodului curent.

c. Căutarea se continuă în subarborele drept al nodului curent.

d. Căutarea se întrerupe deoarece nodul curent este un nod frunză (nu există subarbori pentru nodul respectiv).

   În acest caz, înregistrării curente nu-i corespunde nici un nod în arbore; înregistrarea curentă, de obicei, se inserează ca fiu stîng sau drept al nodului la care s-a ajuns.

Se observă că în fiecare nod, procesul de căutare poate continua numai în subarborele stîng sau numai în cel drept al nodului respectiv. În felul acesta, la fiecare pas se exclud de la căutare nodurile unui subarbore.

Amintim că se poate ajunge la o eficienţă slabă, comparabilă cu cea obţinută la utilizarea listelor, dacă arborele degenerează într-o listă (fiecare nod care nu este o frunză are tot timpul numai descendentul stîng sau tot timpul numai descendentul drept).

Degenerarea arborelui se obţine atunci cînd înregistrările se inserează în arbore într-o astfel de ordine încît cheile lor satisfac tot timpul criteriul (sau negaţia acestuia) utilizat la căutarea în arbore. De obicei, această situaţie are o probabilitate mică de apariţie.

O a treia soluţie pentru organizarea tabelelor dinamice o constituie aşa numitele *tabele de dispersie*. În [3] se descrie, în detaliu, utilizarea tabelelor de acest tip.

În capitolul de faţă vom prezenta o variantă simplificată de utilizare a tabelelor de dispersie la construirea *tabelelor de simboluri*.

13.1. Tabela de cuvinte rezervate

O tabelă de cuvinte rezervate este o tabelă ordonată de dimensiune fixă, dinainte cunoscută. Pentru a fixa ideile să presupunem că, cuvintele

rezervate sînt cuvinte împrumutate din limba engleză şi sînt utilizate într-un limbaj de programare, avînd fiecare un înţeles predefinit.

În acest caz cheile, precum şi înregistrările, se reduc fiecare la cuvintele rezervate. Ordinea impusă cheilor va fi cea *alfabetică*.

Tabela poate fi realizată simplu în limbajul C şi anume vom folosi un tablou de pointeri spre caractere şi fiecare element al acestuia se iniţializează cu adresa de început a zonei de memorie în care se păstrează un cuvînt rezervat. De exemplu, dacă avem în vedere cuvintele rezervate din limbajul C, atunci tabela de cuvinte rezervate se poate declara ca mai jos:

```
char *tcr[]={
 "break",
 "case",
 ...
 "while"
};
```

unde cuvintele rezervate au fost scrise în ordine alfabetică.

Procedura de căutare într-o astfel de tabelă poate fi *căutarea binară* care a fost utilizată pentru un tablou cu elemente numerice în exerciţiul 4.42.

În cazul de faţă se schimbă numai modul de realizare al comparaţiilor dintre elementele tabelei şi elementul curent care se caută în tabelă. În exerciţiul amintit mai sus, comparaţia se realizează folosind operatorii relaţionali obişnuiţi (== şi >) deoarece se compară numere. În cazul de faţă, pentru a compara două cuvinte rezervate,se utilizează funcţia de bibliotecă *strcmp*. Funcţia de căutare returnează:

- indicele cuvîntului rezervat dacă acesta a fost găsit în tabelă;
- -1 în caz contrar.

Cu aceste precizări, funcţia de căutare în tabela de cuvinte rezervate se poate defini ca mai jos:

```
int cautrez(char *pcrt)
/*- cauta in tabela de cuvinte rezervate cuvintul spre care pointeaza pcrt;
 - returneaza indicele cuvintului in tabela sau -1 daca nu exista. */
{
 int inf,sup,i,j;

 static char *tcr[]={
     "break",
      "case",
      ...
     "while"
```

```
};

inf=0;
sup=sizeof(tcr)/sizeof(char *)-1;
for(i=(inf+sup)/2;inf <= sup;i=(inf+sup)/2)
   if((j=strcmp(pcrt,tcr[i]))==0)

/* cuvintul cautat s-a gasit in tabela */
        return i;
   else
        if(j > 0)

/* cuvintul care se cauta este dupa cel spre care pointeaza tcr[i],
 deci limita inferioara poate fi marita la i+1 */
            inf=i+1;
        else

/* cuvintul care se cauta este inaintea celui spre care pointeaza tcr[i],
 deci limita superioara poate fi micsorata la i-1 */
            sup=i-1;

/* cuvintul cautat nu a fost gasit in tabela */
 return -1;
}
```

13.2. Tabela de dispersie

Tabelele de dispersie au la bază o funcţie de transformare a cheilor înregistrărilor:

hf:K -> H

unde:

K - Este mulţimea cheilor;

H - Este o mulţime de numere naturale.

În general, funcţia *hf* nu este injectivă, existînd două sau mai multe chei pentru care *hf* are aceeaşi valoare.

Funcţia *hf* se numeşte *funcţie de dispersie* (în engleză hashing), iar utilizarea ei conduce la o metodă numită *tehnică de dispersie* (tehnică de hashing).

Prima problemă care se pune în legătură cu tabela de dispersie este aceea

de a alege funcţia de dispersie.

În primul rînd, se poate presupune că 0 <=hf(k)<M, pentru orice cheie k.

Aceasta se obţine dacă:

hf(k) = f(k) mod M,

unde:

f(k) - Transformă cheia *k* într-un număr natural.

Relaţia de mai sus defineşte pe hf(k), egal cu restul împărţirii întregi a lui f(k) prin *M*. Pentru a obţine o repartizare cît mai uniformă a valorilor funcţiei *hf*, numărul *M* se alege să fie prim (vezi[3]).

Despre două chei *ki* şi *kj* diferite, se spune că intră în *coliziune*, dacă hf(ki)=hf(kj).

O funcţie de dispersie trebuie să satisfacă următoarele condiţii (vezi [3]):

- valoarea ei să se calculeze simplu şi rapid;
- să minimizeze numărul coliziunilor.

Numărul coliziunilor este dependent de cheile înregistrărilor, precum şi de funcţia *hf*. Cu cît aceasta realizează o repartiţie mai uniformă a valorilor cheilor, cu atît numărul coliziunilor va fi mai mic.

Numărul coliziunilor poate creşte dacă numărul *M* este prea mic. Pe de altă parte, creşterea exagerată a lui *M* conduce la un consum mare de memorie. De aceea, *M* trebuie ales printr-un compromis între necesarul de memorie şi timpul de căutare suplimentar cheltuit pentru rezolvarea coliziunilor.

Funcţia f(k) se alege în funcţie de natura cheilor. Dacă *k* este o cheie numerică, atunci se poate alege f(k)=k.

În cazul în care cheile nu sînt numerice, atunci există mai multe posibilităţi de a transforma cheile în numere naturale.

Aşa de exemplu, se poate considera o codificare numerică a caracterelor unei astfel de chei şi prin aceasta fiecare cheie se transformă într-un şir de coduri binare, iar acesta din urmă poate fi interpretat ca reprezentînd un întreg binar fără semn. O astfel de metodă, deşi pare simplă, poate conduce la calcule greoaie cu numere mari. De aceea se adoptă metode mai simple. Aşa de exemplu, se poate proceda la a păstra numai primele 2-3 coduri binare din reprezentările cheilor sau numai ultimele 2-3 coduri sau un număr mic de coduri din mijlocul cheilor.

O metodă simplă, aplicată frecvent, este aceea de a însuma codurile

caracterelor din compunerea unei chei. Mai jos, vom utiliza şi noi acest procedeu simplu pentru a transforma cheile în numere naturale.

Exemplu:

Fie M = 127. Presupunem că dorim să construim o tabelă de simboluri. În acest caz cheile înregistrărilor sînt identificatori (succesiuni de litere şi eventual şi cifre care, de obicei, încep cu o literă). Vom presupune că se utilizează codul ASCII pentru reprezentarea caracterelor. Fie identificatorul AB1. Atunci

f(AB1) = 'A'+'B'+'1' = 65+66+49 = 180
hf(AB1) = 180%127 = 53.

Evident, identificatorii A1B, BA1, B1A au aceeaşi dispersie, deoarece:

f(AB1) = f(A1B) = f(BA1) = f(B1A) = 180

Deci, toţi identificatorii de mai sus intră în coliziune.

Problemele implicate de coliziune pot fi rezolvate în mai multe moduri (vezi [3]).

În cartea de faţă adoptăm o soluţie simplă şi anume aceea de a rezolva coliziunile cu ajutorul listelor simplu înlănţuite. Astfel, toate înregistrările pentru care cheile au aceeaşi dispersie se inserează într-o listă simplu înlănţuită.

Vom avea deci, mai multe liste simplu înlănţuite, fiecare listă conţinînd înregistrările ale căror chei au aceeaşi dispersie.

Pentru a putea gestiona înregistrările din aceste liste este necesar să cunoaştem pointerul spre primul nod al fiecărei astfel de liste.

Numărul maxim al acestor liste este *M*, deoarece funcţia hf are *M* valori: 0,1,...,M-1. Pointerii spre începuturile listelor respective se păstrează într-un tablou de *M* elemente. Numim *thash* acest tablou. Atunci *thash[i]* are ca valoare pointerul spre începutul listei ce conţine înregistrările pentru care cheile au dispersia egală cu *i*.

Reluînd exemplul de mai sus, *thash[53]* are ca valoare pointerul spre începutul listei care conţine înregistrările cu cheile: AB1, BA1 şi B1A.

Iniţial, aceste liste nu există şi în consecinţă elementele *thash[i]* pentru i=0,1,2,...,M-1 au valoarea zero.

În capitolul 11 s-au utilizat variabilele globale *prim* şi *ultim* pentru a defini capetele unei liste simplu înlănţuite. În cazul de faţă un element al tabloului *thash* joacă acelaşi rol ca şi variabila *prim*. Date fiind operaţiile avute în vedere asupra listelor care conţin înregistrările aflate în coliziune,

se constată că nu avem nevoie de variabila *ultim*.

Procedura de căutare într-o tabelă de dispersie se realizează în următorii paşi:

1. Dacă *thash[h]=0*, unde *h* este dispersia cheii curente, atunci înregistrarea respectivă nu există în tabelă şi funcţia de căutare returnează valoarea zero.

2. Dacă *thash[h]* este diferit de zero, atunci acesta este pointerul spre începutul listei în care se află înregistrările pentru care cheile au dispersia egală cu *h*.

   Căutarea înregistrării curente se face în această listă începînd cu nodul spre care pointează *thash[h]*.

   Această căutare se realizează secvenţial, nod după nod, pînă cînd se găseşte o înregistrare cu aceeaşi cheie ca şi cea curentă sau pînă cînd se ajunge la ultimul nod al listei, lucru care se întîmplă cînd înregistrarea nu este prezentă în lista respectivă.

   În primul caz funcţia de căutare returnează pointerul spre înregistrarea găsită, iar în al doilea caz returnează valoarea zero.

Din cele de mai sus rezultă că procedura de căutare este cu atît mai eficientă, cu cît listele cu înregistrările aflate în coliziune sînt mai mici. Aceasta este funcţie de constanta *M* şi de funcţiile *hf* şi *f* definite mai sus.

De obicei, în cazul tabelelor de simboluri utilizate în compilatoare, *M* se alege sub 1000. De asemenea, însumarea codurilor caracterelor identificatorilor s-a dovedit a fi o soluţie simplă şi eficientă pentru calculul dispersiei. În acest caz funcţia *hf* de calcul a dispersiei se defineşte ca mai jos:

```
#define M 127
unsigned hf(char *s)
{
 unsigned h;
 for(h = 0;*s;)
     h += *s++;
 return h%M;
}
```

În continuare, definim funcţia de căutare în tabela de dispersie, presupunînd că aceste chei sînt şiruri de caractere.

Numim *hcaut* această funcţie. Ea returnează pointerul spre înregistrarea căutată sau zero dacă înregistrarea respectivă nu există în tabelă.

O înregistrare are tipul TNOD definit ca în cazul listelor simplu

înlănţuite:

typedef struct *tnod* {
declaratii
char **cheie*;
declaratii
struct *tnod* **urm*;
} *TNOD*;

Tabloul *thash* are *M* elemente şi elementele lui sînt pointeri spre tipul TNOD. Vom presupune că acest tablou este global.

Iniţial elementele tabloului *thash* au valoarea zero (toate listele sînt vide). Acest tablou se defineşte astfel:

TNOD *thash[M];

Funcţia *hcaut* are ca parametri pointerul spre şirul de caractere din compunerea cheii (se caută înregistrarea a cărei cheie coincide cu acest şir de caractere) şi valoarea dispersiei acestui şir.

Cu aceste precizări, funcţia de căutare se defineşte astfel:

```
TNOD *hcaut(char *s,unsigned h)
/* - cauta inregistrarea pentru care cheie pointeaza spre un sir de
     caractere identic cu cel spre care pointeaza s;
   - returneaza pointerul spre inregistrarea respectiva sau zero daca
     nu exista o astfel de inregistrare. */
{
 TNOD *p;

/* cauta inregistrarea in lista spre care pointeaza thash[h] */
 for(p = thash[h];p;p = p -> urm)
     if(strcmp(s,p -> cheie)==0)

/* s-a gasit inregistrarea cautata */
          return p;

/* nu exista o inregistrare pentru care cheie sa pointeze spre un sir
   identic cu cel spre care pointeaza s */
 return 0;
}
```

Operaţia de introducere a unei înregistrări într-o tabelă de simboluri se va realiza printr-o funcţie pe care o numim *intab*. Ca şi în cazul listelor, funcţia *intab* va utiliza funcţiile *incnod* şi *elibnod*, prima pentru a încărca datele înregistrării în zona de memorie alocată prin funcţia *malloc*, iar a

doua pentru a elibera o astfel de zonă.

Deoarece tabloul *thash* conţine pointeri spre începutul listelor şi nu spre sfîrşitul lor, va fi util ca operaţia de inserare a nodului nou în listă să se facă înaintea primului ei nod, adică folosind o funcţie analogă cu funcţia *iniprim* definită în paragraful 11.1.3.1.

Astfel, dacă *p* pointează spre nodul care se inserează în listă, iar *h* este dispersia cheii lui, atunci ştim că *thash[h]* pointează spre primul nod al listei în care urmează să se insereze înregistrarea spre care pointează *p*. Cum înregistrarea se inserează înaintea primului nod al listei, va trebui ca după inserare p -> urm să pointeze spre acest prim nod, deci se face atribuirea:

p -> urm = thash[h].

Totodată, înregistrarea care se inserează devine primul nod al aceleaşi liste, deci în continuare va trebui ca:

thash[h] = p.

În felul acesta, operaţia de inserare a înregistrării curente se realizează conform paşilor de mai jos:

1. Rezervă zonă pentru înregistrarea curentă folosind funcţia *malloc*.
2. Încarcă datele în zona rezervată la punctul 1.
3. Caută înregistrarea curentă în tabelă.

   Dacă există, atunci se revine din funcţie returnîndu-se pointerul spre înregistrarea găsită.

   Altfel se inserează înregistrarea curentă în tabelă şi se returnează pointerul spre ea.

Funcţia returnează valoarea zero dacă nu mai există înregistrări sau s-a depistat o eroare la încărcarea datelor în înregistrarea curentă.

Definim mai jos funcţia *intab*.

```
TNOD *intab()
/* - determina inregistrarea curenta si o insereaza daca nu a fost gasita
     in tabela;
   - returneaza pointerul spre inregistrarea respectiva sau NULL in
     cazul cind nu mai exista inregistrari sau la eroare. */
 TNOD *p,*q;
 unsigned h;

/* rezerva zona si incarca datele inregistrarii curente */
 if(((p=(TNOD *)malloc(sizeof(TNOD)))!=0)
```

```
        &&(incnod(p)==1)){

/* calculeaza dispersia cheii */
        h = hf(p -> cheie);

/* - daca thash[h]!=0, atunci exista in tabela inregistrari a caror chei
   au dispersia h;
 - in acest caz se cauta inregistrarea in tabela. */
        if(thash[h])
            if((q=hcaut(p -> cheie,h))!=0)

/* inregistrarea exista in tabela */
                return echivalenta(q,p);
            else{

/* - inregistrarea nu exista in tabela;
 - se insereaza inaintea nodului spre care pointeaza thash[h]. */
                p -> urm = thash[h];
                thash[h] = p;
                return p;
            }
        else{

/* nu exista inregistrari care sa intre in coliziune cu inregistrarea
   curenta */
            thash[h] = p;
            p -> urm = 0;
            return p;
        }
    }
    if(p==0){
        printf("memorie insuficienta\n");
        exit(1);
    }

/* eroare la incarcarea datelor sau nu mai sînt inregistrari */
    elibnod(p);
    return 0;
}
```

Funcţia *echivalenta* se apelează în cazul în care în tabelă există o înregistrare de aceeaşi cheie cu cea a înregistrării curente. În general, această funcţie are ca efect modificarea sau actualizarea datelor înregistrării din tabelă cu cele din înregistrarea curentă.

Ea a fost utilizată şi în cazul inserării nodurilor în arbori binari (vezi capitolul 12).

În cazul în care înregistrarea curentă nu conţine date pentru modificarea înregistrării din tabelă care-i corespunde, funcţia *echivalenta* se reduce la ştergerea înregistrării curente, ca mai jos:

```
TNOD *echivalenta(TNOD *q,TNOD *p)
/* sterge inregistrarea spre care pointeaza p si returneaza
  valoarea lui q */
{
 elibnod(p);
 return q;
}
```

Observaţii:

1. Sistemul format din listele care conţin înregistrările aflate în coliziune şi din tabloul de pointeri spre primele înregistrări ale acestor liste, constituie o *tabelă de dispersie*.

   La baza inserării şi accesului la o înregistrare din tabela de dispersie se află funcţia *hf* de calcul a dispersiei.

2. Tabela de dispersie poate fi organizată înlocuind listele cu înregistrările aflate în coliziune prin arbori binari.

   În acest caz, elementele tabloului *thash* au ca valori pointeri spre rădăcinile acestor arbori.

   În felul acesta operaţiile de căutare sînt, de obicei, mai performante (excepţie fac cazurile cînd arborii sînt degeneraţi, dar această posibilitate este extrem de redusă).

   În felul acesta, un arbore mare s-a descompus în mai mulţi (cel mult *M*) care sînt relativ mici. Astfel, căutarea unei înregistrări nu se mai face într-un arbore mare ci într-unul mult mai mic care a fost selectat pe baza unui calcul simplu.

Exerciţii

13.1 Să se scrie o funcţie care calculează dispersia unui şir de caractere prin însumarea codurilor caracterelor şirului respectiv.

Aceasta este funcţia *hf* definită în paragraful 13.2.

FUNCŢIA BXIII1

```
unsigned hf(char *s) /* calculeaza dispersia sirului spre care
```

pointeaza s */

```
{
 unsigned h;

 for(h=0; *s;)
     h += *s++;
 return h%M;
}
```

Observaţie:

M este o constantă simbolică şi reprezintă numărul de elemente al tabloului de pointeri *thash*.

13.2 Să se scrie funcţia *hcaut* care caută o înregistrare de cheie dată într-o tabelă de dispersie.

Funcţia de faţă a fost definită în paragraful 13.2.

FUNCŢIA BXIII2

```
TNOD *hcaut(char *s,unsigned h)
/* - cauta inregistrarea de cheie identica cu sirul spre care pointeaza s
     si a carei dispersie este h;
   - returneaza pointerul spre inregistrarea respectiva sau zero
     daca inregistrarea este absenta. */
{
 TNOD *p;

/* cauta inregistrarea in lista spre care pointeaza thash[h] */
 for(p=thash[h];p;p=p -> urm)
     if(strcmp(s,p -> cuvant)==0)

/* s-a gasit inregistrarea curenta */
         return p;

/* nu exista inregistrarea cautata */
 return 0;
}
```

13.3 Să se scrie funcţia *intab* care inserează o înregistrare într-o tabelă de dispersie.

Funcţia a fost definită în paragraful 13.2.

FUNCȚIA BXIII3

```
TNOD *intab()
/* - determina inregistrarea curenta si o insereaza daca nu exista in tabela
     de dispersie;
   - returneaza pointerul spre inregistrarea respectiva sau zero in
     cazul cind nu mai exista inregistrari sau la eroare. */
{
 TNOD *p,*q;
 unsigned h;

/* rezerva zona si incarca datele inregistrarii */
 if((p=(TNOD *)malloc(sizeof(TNOD))) &&
    (incnod(p)==1)){

/* calculeaza dispersia cheii inregistrarii curente */
      h=hf(p -> cuvant);

/* daca thash[h]!=0, se cauta inregistrarea in tabela */
      if(thash[h])
           if(q=hcaut(p -> cuvant,h))

/* s-a gasit inregistrarea cautata */
                return echivalenta(q,p);
           else{

/* - inregistrarea curenta nu exista in tabela;
   - se insereaza */
                p -> urm=thash[h];
                thash[h]=p;
                return p;
           }
      else{

/* nu exista inregistrari care sa intre in coliziune cu inregistrarea
   curenta */
           p -> urm =0;
           thash[h]=p;
           return p;
      }
 }
 if(p==0){
     printf("memorie insuficienta\n");
     exit(1);
```

```
 }
 elibnod(p);
 return 0;
}
```

13.4 Să se scrie un program care citeşte cuvintele unui text şi scrie numărul de apariţii al fiecăruia dintre ele.

Această problemă a fost rezolvată în capitolele precedente folosind:

- listele simplu înlănţuite;
- listele dublu înlănţuite;
- arborii binari.

În exerciţiul de faţă vom folosi o tabelă de dispersie. Problema, ca şi în cazul utilizării celorlalte metode, se rezolvă conform paşilor de mai jos:

1. Se citeşte cuvîntul curent;
2. Se caută cuvîntul curent în tabela de dispersie.

   Dacă el există deja în tabelă, atunci se incrementează numărătorul său.

   Altfel se inserează o înregistrare în tabela corespunzătoare cuvîntului curent citit şi se iniţializează numărătorul lui cu valoarea 1.

Aceşti paşi se execută pînă la întîlnirea sfîrşitului de fişier care marchează sfîrşitul textului.

În programul de faţă se folosesc unele funcţii care s-au utilizat şi în programele precedente. Aceste funcţii sînt:

citcuv, incnod, elibnod şi *echivalenta*.

PROGRAMUL BXIII4

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <alloc.h>
#include <string.h>

typedef struct tnod {
 char *cuvant;
 int frecventa;
 struct tnod *urm;
} TNOD;
```

```
#define M 127

TNOD *thash[M];

#include "BX48.CPP"    /* citcuv      */
#include "BXI1.CPP"    /* incnod      */
#include "BXI2.CPP"    /* elibnod     */
#include "BXII1.CPP"   /* echivalenta */
#include "BXIII1.CPP"  /* hf          */
#include "BXIII2.CPP"  /* hcaut       */
#include "BXIII3.CPP"  /* intab       */

main() /* citeste un text si afiseaza frecventa cuvintelor citite */
{
 TNOD *p;
 int i,j;

/* initializeaza tabloul thash cu pointerul nul */
 for(i=0;i < M; i++)
     thash[i]=0;

/* - creeaza tabela de dispersie;
   - fiecarui cuvint citit din textul sursa ii corespunde o inregistrare
     de tipul TNOD. */
 while(intab())
     ;

/* se listeaza cuvintele citite si frecventa lor */
 j=0;
 for(i=0;i < M;i++)
     for(p=thash[i];p;j++,p=p -> urm){
         printf("cuvintul: %-51s are\
                frecventa=%d\n", p -> cuvant,
                                 p -> frecventa);
         if((j+1)%23==0){
            printf("Actionati o tasta pentru a\
                  continua\n");
            getch();
          }
 }
}
```

13.5 Să se scrie un program care citeşte cuvintele dintr-un text şi afişează

următoarele date despre fiecare cuvînt distinct citit:

- caracterele cuvîntului;
- dacă este rezervat sau nu;
- numărul liniei în care apare prima dată cuvîntul.
- numărul coloanei în care începe cuvîntul în linia afişată;
- numărul de apariţii al cuvîntului în textul respectiv.

Cuvintele care se citesc se vor păstra într-o tabelă de dispersie. Înregistrările acestei tabele sînt de tipul TNOD definit mai jos:

typedef struct *tnod* {
    char **cuvant*;
    int *frecventa*;
    int *nr_linie*;
    int *nr_col*;
    Boolean *tip*;
} *TNOD*;

Variabila *tip* are valoarea 1 dacă cuvîntul este rezervat şi 0 în caz contrar.

În program se defineşte o tabelă de cuvinte rezervate. Acestea sînt alese din setul de cuvinte rezervate ale limbajului C.

Programul se realizează în conformitate cu paşii de mai jos:

1. Se citeşte un cuvînt de la intrare.

   Dacă s-a întîlnit sfîrşitul de fişier se trece la pasul 5.

   Altfel se continuă cu pasul 2.

2. Se construieşte înregistrarea de tip TNOD pentru cuvîntul citit corect.
3. Se caută înregistrarea curentă în tabela de dispersie.

   Dacă există, se incrementează frecvenţa cuvîntului din înregistrarea găsită în tabela de dispersie.

   Altfel se inserează înregistrarea în tabelă.

4. Se trece la pasul 1.
5. Se listează datele păstrate în tabela de dispersie.

Prin cuvînt se înţelege o succesiune de litere mici şi/sau mari.

PROGRAMUL BXIII5

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <conio.h>
#include <alloc.h>
#include <string.h>

typedef enum {False,True} Boolean;

typedef struct tnod{
 char *cuvant;
 int frecventa;
 int nr_lin;
 int nr_col;
 Boolean tip;
 struct tnod *urm;
} TNOD;

#define M 127

TNOD *thash[M];
int linie,coloana;

int incnod(TNOD *p) /* citeste cuvintul curent si construieste
                       inregistrarea corespunzatoare lui */
{
 int c,i,j;
 char t[255];
 char *q;
 int inf,sup;
 static char *tcr[]={
     "break","case","char","continue","default",
     "do","double",  "else","extern","float","for",
     "goto","if","int","long","register","return",
     "short","sizeof","static","struct","switch",
     "typedef","union","unsigned","void","while"
 };

 inf=0;
 sup=sizeof(tcr)/sizeof(char *)-1;
 p -> cuvant=0;
 while((c=getchar())!=EOF){
     coloana++;
     if(c >= 'A' && c <= 'Z' || c >= 'a' &&
        c <= 'z')
```

```
            break;
         if(c=='\n'){
            linie++;
            coloana=0;
        }
    }
    if(c==EOF)
        return -1;
    p -> nr_lin = linie;
    p -> nr_col = coloana;
    for(i=0;i < 255;i++){
        if(!(c >= 'A' && c <= 'Z') && !(c >= 'a' &&
             c <= 'z')){
            if(c=='\n'){
                linie++;
                coloana=0;
            }
            break;
        }
        t[i]=c;
        c=getchar();
        coloana++;
    }
    if(c==EOF)
        return -1;
    t[i++]='\0';
    if((q=(char *)malloc(i))==0){
        printf("memorie insuficienta\n");
        exit(1);
    }
    strcpy(q,t);
    p -> cuvant=q;
    p -> frecventa=1;
```

/* cauta cuvintul citit in tabela de cuvinte rezervate */

```
    for(i=(inf+sup)/2;inf <= sup;i=(inf+sup)/2){
        if((j=strcmp(t,tcr[i]))==0)
            break;
        if(j < 0)
            sup=i-1;
        else
            inf=i+1;
    }
```

```
 if(j)

/* nu este cuvint rezervat */
     p -> tip = False;
 else /* cuvint rezervat */
     p -> tip = True;
 return 1;
} /* sfirsit incnod */

#include "BXI2.CPP"    /* elibnod     */
#include "BXII11.CPP"  /* echivalenta */
#include "BXIII1.CPP"  /* hf          */
#include "BXIII2.CPP"  /* hcaut       */
#include "BXIII3.CPP"  /* intab       */

main() /* - citeste cuvintele unui text;
          - afiseaza pentru fiecare cuvint distinct din text:
               - caracterele cuvintului;
               - daca este rezervat sau nu;
               - numarul liniei si coloanei in care apare cuvintul
                 de prima data;
               - numarul de aparitii al cuvintului respectiv in textul
                 citit. */
{
 int i,j;
 TNOD *p;

/* initializari */
 linie = 1;
 coloana = 0;
 for(i = 0;i < M;i++)
     thash[i] = 0;

/* se citesc cuvintele de la intrare si se construieste tabela de dispersie */
 while(intab())
     ;

/* afiseaza datele din tabela de dispersie */
 j = 0;
 for(i = 0;i < M;i++)
     for(p = thash[i];p;j++,p = p -> urm){
         printf("\t%s\n",p -> cuvant);
         if(p -> tip == True)
```

```
            printf("  rezervat");
        else
            printf("nerezervat");
        printf(" linia:%6d\t coloana=%6d\t\
              aparitii:%d\n", p -> nr_lin,
               p -> nr_col, p -> frecventa);
        if((j+1)%10==0){
            printf("Actionati o tasta pentru a\
                    continua\n");
            getch();
        }
    }
} /* sfirsit main */
```

14. SORTARE

În prezent există mai multe metode de sortare a datelor. Prin *sortare* înţelegem aranjarea datelor într-o anumită ordine. În acest scop considerăm că datele sînt o colecţie de elemente de un anumit tip şi fiecare element conţine o dată sau chiar mai multe, în raport cu care se realizează ordonarea. O astfel de dată se numeşte *cheie*.

Sortarea se poate realiza:

1. Fie aranjînd datele care se sortează în aşa fel încît cheile lor să corespundă ordinii dorite.
2. Fie ordonînd un tablou de pointeri spre datele care trebuiesc sortate în aşa fel încît considerîndu-i pe aceştia în ordinea crescătoare a indicilor tabloului, datele spre care pointează să formeze o mulţime ordonată în conformitate cu ordinea dorită.

Un exemplu de sortare, de felul celui indicat la punctul 1 de mai sus, este exerciţiul 4.40. În acest exemplu se defineşte o funcţie care ordonează crescător elementele unui tablou de tip *double*.

Exerciţiul 8.14. este un exemplu de sortare folosind un tablou de pointeri aşa cum se indică la punctul 2 de mai sus. În acest caz se sortează un şir de cuvinte ordonînd pointerii care pointează spre ele. Aceştia sînt păstraţi într-un tablou de pointeri spre caractere. În timpul sortării se fac permutări ale acestor pointeri în aşa fel încît, în final, aceşti pointeri dacă sînt consideraţi în ordinea crescătoare a indicilor, atunci cuvintele spre care pointează se află în ordine alfabetică.

În ambele cazuri metoda de sortare a fost aceeaşi, metoda bulelor. De aceea, cele două cazuri de mai sus nu trebuiesc considerate ca fiind metode de sortare diferite.

Metoda bulelor are la bază compararea a două chei învecinate (de indici succesivi sau din noduri succesive). Dacă acestea nu sînt în ordinea cerută, atunci se permută elementele respective sau pointerii spre ele. O caracteristică a acestui algoritm este faptul că la fiecare parcurgere a şirului de date care se sortează, cel puţin un element ajunge să-şi ocupe poziţia sa finală în conformitate cu ordonarea cerută. În cazul exerciţiilor amintite mai sus nu s-a ţinut seama de această proprietate. Propunem cititorului să rescrie exerciţiile respective ţinînd seama de faptul că la fiecare parcurgere cel puţin un element bine determinat este poziţionat pe locul lui final. Aceasta înseamnă că la fiecare parcurgere se poate diminua cu 1 numărul elementelor de analizat în următoarea parcurgere. Aceasta conduce la o

îmbunătăţire a eficienţei algoritmului de sortare prin metoda bulelor. Cu toate acestea, această metodă rămîne printre cele cu cel mai mare număr de permutări.

O altă metodă de sortare este legată de folosirea *arborilor binari*. Datele care se sortează se inserează în nodurile unui arbore binar şi apoi acesta se listează în *inordine*. O astfel de metodă a fost utilizată în exerciţiul 12.14. Programul definit în acest exerciţiu citeşte cuvintele dintr-un text şi apoi le afişează în ordine alfabetică.

În general, sortarea cu ajutorul *arborilor binari* este mai eficientă decît metoda *bulelor*.

Există diverşi algoritmi de sortare bazaţi pe arbori. Pentru detalii cititorul poate consulta [3].

În capitolul de faţă prezentăm două metode de sortare utilizate frecvent în diferite aplicaţii. Aceste metode sînt relativ simple şi în acelaşi timp asigură o eficienţă mai bună decît metoda bulelor.

Una dintre aceste metode este cunoscută sub denumirea de *sortare Shell* sau *sortare cu micşorarea incrementului*.

Cealaltă metodă se numeşte *quick sort* (sortare rapidă).

În continuare, vom avea în vedere numai sortări cu chei numerice întregi. Aceasta nu este o restricţie esenţială, deoarece sortările cu chei nenumerice se realizează similar, folosind tablouri de pointeri.

14.1. Sortare Shell

Metoda *Shell* a fost publicată de către Donald L. Shell, în lucrarea [19]. Metoda porneşte de la analiza critică a metodei bulelor. În metoda bulelor, se compară elementele vecine şi dacă nu sînt în ordinea cerută, atunci ele se permută. De aceea, elementele care nu sînt în ordine corectă se deplasează cu o singură poziţie. Pentru a îmbunătăţi acest procedeu, ar trebui să realizăm deplasări cu mai multe locuri ale elementelor, la o permutare a lor. Acest lucru se poate realiza dacă în loc de a compara elemente învecinate, comparăm elemente aflate la o anumită distanţă între ele. În cazul în care ele nu sînt în ordinea cerută, ele se vor permuta. În felul acesta elementele se deplasează făcînd salturi mai mari decît o poziţie.

Distanţa dintre elementele comparate se numeşte *increment*. Incrementul se micşorează după o parcurgere a şirului de elemente care se sortează şi se reia parcurgerea de la începutul şirului. De aici rezultă denumirea *sortare cu micşorarea incrementului*.

Înainte de a defini exact metoda de sortare a lui Shell, vom considera un exemplu.

Fie de sortat, în ordine crescătoare, şirul de numere:

173 25 4 300 256 83 95 14 415 3

La început trebuie ales incrementul.

O metodă simplă, dar nu chiar cea mai eficientă, este de a fixa valoarea acestuia la jumătate din numărul total al elementelor de sortat. În cazul de faţă incrementul va fi:

inc = 5

Apoi, incrementul se înjumătăţeşte după fiecare parcurgere a şirului. Algoritmul se termină cînd inc = 0.

Algoritmul începe cu a compara primul element al şirului (173) cu al 6-lea (incrementul fiind 5) (83). Cum 173 > 83, se face permutarea elementelor respective:

83 25 4 300 256 173 95 14 415 3

Compararea următoare se realizează pentru elementul al 2-lea cu al 7-lea, deci se compară 25 cu 95. Ele fiind în ordinea cerută, nu se permută.

Apoi se compară elementele următoare, adică 4 cu 14. Ele fiind în ordinea cerută nu se permută. În continuare se compară elementele următoare, adică 300 cu 415, care şi ele sînt în ordinea cerută şi deci nu se permută. Elementele următoare 256 şi 3 nu sînt în ordine crescătoare şi în consecinţă ele se permută. Se obţine:

83 25 4 300 3 173 95 14 415 256

Se înjumătăţeşte incrementul şi se obţine:

inc = 2

Se reia parcurgerea şirului ca mai jos.

Se compară primul element (83) cu al treilea (4). Deoarece 83 > 4, se permută elementele respective:

4 25 83 300 3 173 95 14 415 256

Apoi se compară al doilea element cu al patrulea, adică 25 cu 300. Acestea sînt în ordine crescătoare şi de aceea ele nu se permută. Apoi se compară elementul al treilea cu al cincilea, adică 83 cu 3. Cum 83 > 3, elementele respective se permută şi se obţine:

4 25 3 300 83 173 95 14 415 256

În acest moment se permută şi primul element cu al treilea, deoarece nici acestea nu se află în ordinea cerută. Se obţine:

3 25 4 300 83 173 95 14 415 256

La fiecare permutare a două valori, algoritmul compară valoarea mai mică permutată cu cea precedentă (poziţia precedentă se stabileşte folosind incrementul) şi realizează o nouă permutare dacă este nevoie.

Această comparaţie se continuă, propagîndu-se spre începutul şirului, pînă cînd:

a. nu se mai fac permutări;

sau

b. nu mai există elemente în şir (se ajunge la un element care nu mai are precedent în şir).

În continuare se compară elementul al patrulea cu al şaselea, adică 300 cu 173. Acestea se permută:

3 25 4 173 83 300 95 14 415 256

Apoi se compară elementul mai mic permutat (173) cu precedentul lui, conform incrementului, adică cu 25. Cum 25 < 173, aceste elemente nu se permută.

Se continuă parcurgerea şirului, adică se compară elementul al 5-lea cu al 7-lea (83 şi 95). Acestea sînt în ordine crescătoare şi deci nu se permută. Apoi se continuă cu elementul al 6-lea, care se compară cu al 8-lea: 300 cu 14. Acestea se permută şi se obţine:

3 25 4 173 83 14 95 300 415 256

Apoi se compară 173 cu 14 şi se constată că şi aceste elemente trebuie să se permute:

3 25 4 14 83 173 95 300 415 256

Propagarea comparării spre începutul şirului continuă cu încă un pas şi anume, se compară 25 cu 14. Cum 25 > 14 se permută şi aceste elemente obţinîndu-se:

3 14 4 25 83 173 95 300 415 256

Se continuă cu parcurgerea şirului, comparîndu-se elementul al 7-lea cu al 9-lea. Aceste elemente sînt 95 şi 415, care sînt în ordine crescătoare, deci ele nu se permută. În sfîrşit, se compară elementul al 8-lea cu al 10-lea şi se constată că ele trebuie permutate. Se obţine:

3 14 4 25 83 173 95 256 415 300

Compararea elementului al 8-lea cu al 6-lea nu conduce la alte permutări şi în felul acesta se încheie cea de a doua parcurgere a şirului. Se înjumătăţeşte incrementul şi se obţine:

inc = 1.

În acest moment se parcurge şirul comparînd elementele învecinate.

Se compară primul element cu al doilea: 3 cu 14. Ei sînt în ordine crescătoare, deci nu se permută. Se compară 14 cu 4 şi cum aceste elemente nu sînt în ordine crescătoare, ele se permută:

3 4 14 25 83 173 95 256 415 300

Apoi se compară elementul al doilea cu precedentul (4 cu 3). Ele sînt în ordinea cerută şi nu se mai permută.

O nouă permutare are loc la compararea elementului al 6-lea (173) cu al 7-lea (95). Se obţine:

3 4 14 25 83 95 173 256 415 300

Nici această permutare nu se propagă spre începutul şirului deoarece 83 < 95.

Ultima permutare se realizează la compararea ultimelor două elemente: 415 cu 300. După permutarea lor se încheie cea de treia parcurgere a şirului:

3 4 14 25 83 95 173 256 300 415

Se înjumătăţeşte incrementul şi se obţine inc = 0. Aceasta corespunde momentului în care şirul este sortat.

Aplicînd metoda bulelor la acelaşi exemplu, se constată că sînt necesare mai multe parcurgeri şi permutări decît în cazul utilizării metodei Shell.

Eficienţa metodei Shell este mult mai mare în comparaţie cu metoda bulelor în cazul în care se sortează un număr mai mare de date (de ordinul sutelor sau miilor).

O influenţă asupra eficienţei metodei o are şi modul în care se alege incrementul. În lucrarea [11] se arată că valorile incrementului la diferite

parcurgeri ale şirului de sortat este bine să fie numere prime între ele.

Pentru alte detalii în legătură cu alegerea incrementului la fiecare pas, vezi [3].

În capitolul de faţă ne mărginim la alegerea incrementului inţial egal cu n/2, unde *n* este numărul elementelor şirului de sortat. De asemenea, după fiecare parcurgere a şirului, acesta se va înjumătăţi. Cazul cel mai nefavorabil al acestei metode se obţine cînd *n* este o putere a lui doi. Acest caz se poate evita dacă la fiecare parcurgere a şirului incrementul se determină prin relaţia:

increment = increment/2 + 1, dacă increment > 2.

Metoda de sortare Shell poate fi definită astfel:

1. inc = n/2, unde prin *n* am notat numărul elementelor care se sortează.
2. Se realizează o parcurgere a şirului de elemente care se sortează.
3. inc = inc/2.
4. Dacă inc > 0, atunci se reia de la pasul 2.

   Altfel şirul este sortat.

Parcurgerea şirului de elemente implică paşii următori:

1. i = inc.
2. j = i-inc+1.
3. Dacă j>0 şi elementele de ordine j şi j+inc nu satisfac criteriul de ordonare, atunci elementele respective se permută.

   Altfel se continuă cu pasul 6.
4. j = j-inc.
5. Se reia de la pasul 3.
6. i = i+1.
7. Dacă i > n, se termină parcurgerea curentă a şirului.

   Altfel se reia de la pasul 2.

Paşii 4 şi 5 se realizează în cazul în care s-a efectuat permutarea elementelor de ordinul j şi j+inc. În acest caz se permută, dacă este necesar, elementele precedente elementului deplasat în poziţia *j*. Elementele precedente sînt:

j-inc, j-2*inc, ...

Aceasta se realizează repetînd pasul 3 atît timp cît j > 0 şi se fac permutări.

Procedînd ca în [2], metoda descrisă mai sus se transcrie imediat în limbajul C, ca mai jos.

Presupunem că elementele de sortat se află într-un tablou *tab* de tip *int* şi ele sînt în număr de *n*.

```
void shellsort(int tab[],int n)
/* sorteaza in ordine crescatoare elementele lui tab prin metoda Shell */
{
 int inc;
 int i,j,t;

 for(inc = n/2;inc > 0; inc /= 2)

/* se realizeaza o parcurgere a sirului */
     for(i = inc;i < n;i++)

/* - se compara doua elemente aflate la distanta inc unul de altul si daca
    este cazul se permuta;
   - daca s-a realizat o permutare, atunci elementul deplasat in fata
    se compara cu precedentele lui si se permuta daca este cazul. */
          for(j = i-inc;j >= 0 &&
                 tab[j] > tab[j+inc];j -= inc){
               t = tab[j];
               tab[j] = tab[j+inc];
               tab[j+inc] = t;
          }
}
```

14.2. Sortare rapidă

Această metodă a fost publicată de C.A.R.Hoare în lucrarea [20].

Ideea sortării rapide, ca şi sortarea lui Shell, se bazează pe faptul că pentru a atinge o viteză mai bună este de dorit ca să permutăm elementele aflate la o distanţă mai mare unul de altul decît 1.

Pentru a ne fixa ideile să presupunem că dorim să sortăm crescător elementele tabloului *tab* de tip *int*, numărul lor fiind *n*.

Alegem, în mod arbitrar, un element al tabloului *tab* şi-l atribuim variabilei *x*. Apoi vom parcurge elementele tabloului şi vom face permutări de elemente dacă este necesar, aşa încît toate elementele mai mici decît *x* să

fie în stînga tabloului (indici mai mici), iar cele mai mari în dreapta lui. În felul acesta, în etapa următoare, se poate considera că avem de sortat două tablouri distincte, unul care conţine elemente mai mici decît *x* şi unul care conţine elemente mai mari decît *x*. Aceste tablouri se sortează şi ele prin acelaşi procedeu. Se poate considera că, acest procedeu constă în divizarea unui tablou în două părţi, aşa încît elementele unuia să fie toate mai mici decît elementele celuilalt. De aceea, această metodă se mai numeşte şi metoda prin *interschimb de poziţii*.

Metoda sortării rapide poate fi realizată ca mai jos:

1. Se alege un element oarecare al tabloului, de exemplu elementul aflat la mijlocul tabloului şi se atribuie lui *x*.
2. Se parcurge tabloul de la stînga la dreapta pînă cînd se găseşte un element tab[i] >= x.
3. Se parcurge tabloul de la dreapta la stînga pînă cînd se găseşte un element tab[j] <= x.
4. Se permută între ele elementele tab[i] cu tab[j], dacă *i* nu-l depăşeşte pe *j*.
5. Se continuă paşii 2, 3 şi 4 de mai sus pînă cînd se ajunge ca i > j.

   Cînd i > j, tabloul este divizat în două.

Paşii de mai sus se continuă apoi, asupra fiecăreia dintre cele două părţi în care s-a divizat tabloul iniţial. În felul acesta se divide fiecare parte în două părţi, realizîndu-se o divizare în 4 părţi a tabloului iniţial şi aşa mai departe. Divizarea continuă pînă cînd fiecare parte conţine un singur element. În acest moment tabloul are elementele sortate în ordine crescătoare.

Evident, se poate defini un algoritm analog pentru a sorta elementele unui tablou în ordine descrescătoare.

Mai jos, exemplificăm metoda sortării rapide folosind tabloul la care s-a aplicat metoda lui Shell:

173 25 4 300 256 83 95 14 415 3

Sortarea se realizează conform paşilor de mai jos:

1. inf = 1 (indicile primului element).
2. sup = 10 (indicele ultimului element).
3. Se alege elementul de indice:

   (inf+sup)/2

deci

x = tab[(1+10)/2] = tab[5] = 256.

4. i =inf.

5. j = sup.

6. Se parcurg elementele tabloului de la stînga la dreapta pînă se găseşte un element mai mare sau egal cu 256.

   Acesta este 300, deci i = 4 şi tab[i] = 300.

7. Se parcurg elementele tabloului de la dreapta spre stînga pînă cînd se găseşte un element mai mic sau egal cu 256.

   Acesta este chiar ultimul element.

   Deci j = 10 şi tab[j] = 3.

8. Cum i < j, se permută tab[i] cu tab[j]:

   173 25 4 3 256 83 95 14 415 300.

9. i = i+1, deci i = 5.

10. j = j-1, deci j = 9.

   Cum i< j, se reia de la punctul 6:

6. Se obţine i = 5, deoarece tab[5] = 256 şi 256 >= 256 este primul element cu această proprietate.

7. Se obţine j = 8, deoarece tab[8] = 14 şi 14 <256 este primul element întîlnit de la dreapta spre stînga care să aibă această proprietate.

8. Cum i < j, se permută cele două elemente:

   173 25 4 3 14 83 95 256 415 300.

9. i = i+1, deci i = 6.

10. j = j-1, deci j = 7.

   Cum i < j se reia de la punctul 6.

6. tab[8] = 256 >= 256 şi acesta este primul element cu această proprietate.

   i = 8.

7. tab[7] = 95 < 256.

   j = 7.

8. Cum i > j, nu se mai fac permutări.

În acest moment s-a terminat procesul de divizare cu i =8 şi j= 7.

Tabloul s-a divizat în două părţi:

- partea întîi: tab[1], tab[2], ..., tab[j = 7];
- partea a doua: tab[8], tab[9] şi tab[10].

În continuare se aplică aceeaşi procedură la fiecare din aceste două părţi pentru prima parte se ia:

inf = 1 şi sup = 7.

La pasul 3 se obţine:

3. x = tab[(1+7)/2] = tab[4] = 3.
4. i = 1.
5. j = 7.
6. tab[1] este primul element aşa încît tab[i] >= x.

   Deci i = 1.
7. tab[4] este primul element aşa încît tab[j] <= x.

   Deci j = 4.
8. Cum i < j, se permută tab[i] cu tab[j]:

   3 25 4 173 14 83 95.
9. i = i+1.

   Deci i = 2.
10. j = j-1.

    Deci j = 3.

    Cum i < j, procesul se reia de la punctul 6.

6. Deoarece tab[i] = 25 > x = 3, *i* nu se modifică.
7. tab[3] = 4, tab[2] = 25 şi tab[1] = 3 deci tab[1] <=x =3 şi j = 1.
8. Cum i = 2 şi j= 1, i > j şi nu se mai fac permutări.

   În acest moment s-a obţinut o nouă divizare.

   Cele două părţi sînt:

   - prima parte: 3;

- a doua parte: 25 4 173 14 83 95.

deoarece prima parte s-a redus la un element, se continuă cu partea a doua:

1. inf = 2.
2. sup = 7.
3. x = tab[(2+7)/2] = tab[4] = 173.
4. i= 2.
5. j = 7.
6. Se obţine i = 4, deoarece tab[4] >= x este primul element cu această proprietate.
7. j = 7, deoarece tab[7] = 95 < x.
8. i < j deci se permută tab[i] cu tab[j]:

   25 4 95 14 83 173.
9. i = i+1.

   Deci i = 5.
10. j = j-1.

    Deci j = 6.

    Cum i < j, se reia de la pasul 6.
6. Se obţine i = 7 deoarece singurul element cu proprietatea tab[i] >= 173 este tab[7] = 173.
7. j nu se modifică deoarece tab[6] = 83 < x = 173.
8. i > j, deci elementele respective nu se permută.

   S-a obţinut o nouă divizare a tabloului:

   - prima parte: 25 4 95 14 83;
   - partea a doua: 173.

   Se continuă cu prima parte:
1. inf = 2.
2. sup = 6.
3. x = tab[(2+6)/2] = tab[4] = 95.
4. i = 2.
5. j = 6.

6. i = 4, deoarece tab[4] = 95 >= 95.

7. j rămîne 6, deoarece tab[6] = 83 < 95.

8. i < j şi de aceea se permută tab[i] cu tab[j]:

   25 4 83 14 95.

9. i = i+1; i = 5.

10. j = j-1; j = 5.

   Cum i <= j se reia de la punctul 6.

6. i = 6 deoarece tab[6] = 95 >= 95.

7. j rămîne 5 deoarece tab[5] = 14 < 95.

8. i > j, deci elementele respective nu se permută.

   S-a obţinut o nouă divizare a tabloului:

   - prima parte: 25 4 83 14;
   - partea a doua: 95.

   Se continuă cu prima parte:

1. inf = 2.

2. sup = 5.

3. x = tab[(2+5)/2] = tab[3] = 4.

4. i = 2.

5. j = 5.

6. i rămîne pe loc deoarece tab[2] = 25 > x = 4.

7. j = 3 deoarece tab[3] = 4 <= x = 4.

8. deoarece i < j, se face permutare:

   4 25 83 14.

9. i = i+1, deci i = 3.

10. j = j-1, deci j = 2.

   Deoarece i > j, s-a obţinut o nouă divizare:

   - prima parte: 4;
   - partea a doua: 25 83 14.

   Se continuă cu partea a doua:

1. inf = 3.
2. sup = 5.
3. x = tab[(3+5)/2] = tab[4] = 83.
4. i = 3.
5. j = 5.
6. i = 4 deoarece tab[4] = 83 >= 83.

   Este primul element cu această proprietate.
7. j rămîne pe loc, deoarece tab[5] = 14 < 83.
8. i < j, deci se face permutarea:

   25 14 83.
9. i = i+1, deci i = 5.
10. j = j-1, deci j = 4.

    Deoarece i > j, s-a obţinut o nouă divizare:
    - prima parte: 25 14;
    - partea a doua: 83.

    Se continuă cu prima parte:

1. inf = 3.
2. sup = 4.
3. x = tab[(3+4)/2] = tab[3] = 25.
4. i = 3.
5. j = 4.
6. i rămîne pe loc deoarece tab[3] = 25 >= x = 25.
7. j rămîne pe loc deoarece tab[4] = 14 <= x = 25.
8. Cum i < j, se face permutare:

   14 25.
9. i = i+1, deci i = 4.
10. j = j-1, deci j = 3.

    Cum i > j, s-a obţinut o nouă divizare:
    - prima parte: 14;

– partea a doua: 25.

În momentul de faţă tabloul *tab* are elementele ordonate ca mai jos:

3 4 14 25 83 95 173 256 415 300.

El este aproape sortat. Partea netratată este partea a doua de la prima divizare:

256 415 300.

Aplicăm acelaşi procedeu:

1. inf = 8.
2. sup = 10.
3. x = tab[(8+10)/2] = tab[9] = 415.
4. i = 8.
5. j = 10.
6. i = 9, deoarece tab[9] = 415 >= x = 415 şi acesta este primul element cu această proprietate.
7. j rămîne pe loc, deoarece tab[10] = 300 < x = 415.
8. i < j, deci se face permutarea:

   256 300 415.

9. i = i+1, i = 10.
10. j = j-1, j = 9.

Cum i > j, s-a obţinut divizarea:

- prima parte: 256 300;
- partea a doua: 415.

Se continuă cu prima parte:

1. inf = 8.
2. sup = 9.
3. x = tab[(8+9)/2] = tab[8] = 256.
4. i = 8.
5. j = 9.
6. i rămîne pe loc, deoarece tab[i] = 256 >= 256.

7. j= 8, deoarece tab[j] = 256 <= 256 şi acesta este primul element cu această proprietate.

8. Cum i = j, nu se face permutare.

9. i = i+1, deci i = 9.

10. j = j-1, deci j = 7.

Cum i > j, procesul se încheie, tabloul *tab* fiind sortat în ordine crescătoare:

3 4 14 25 83 95 173 256 300 415.

Din acest exemplu se observă că diviziunea este în mare măsură dependentă de elementul separator ales la fiecare pas. Dacă acesta este chiar elementul minim dintre elementele tabloului supus divizării, atunci prima parte se reduce chiar la acest element. În mod analog, dacă elementul separator este elementul maxim, atunci partea a doua se reduce chiar la acest element.

Acestea sînt cazurile cele mai ineficiente deoarece tabloul se divide în aşa fel încît una din părţi conţine un singur element.

În general, divizările pentru tablouri mari este bine să fie cît mai echilibrate, adică cele două părţi să fie de lungimi apropriate. Nu există însă un procedeu simplu care să asigure acest fapt. Alegerea elementului de la mijlocul tabloului este o metodă simplă şi destul de practică. Mai jos o să indicăm şi alte metode pentru a alege elementul separator.

În continuare, descriem o funcţie, în limbajul C, pentru metoda ilustrată mai sus. Numim *quicksort* funcţia respectivă. Procedura *quicksort* se descrie simplu dacă funcţia respectivă este recursivă. Natura ei recursivă decurge din faptul că ea constă din următoarele:

- se divide tabloul în două părţi în aşa fel încît elementele unei părţi să nu depăşească elementele celeilalte părţi;
- pentru fiecare parte se aplică aceeaşi procedură de divizare pînă cînd lungimea părţii devine 1.

Tabloul este sortat cînd lungimea fiecărei părţi a devenit egală cu 1.

Procedura de divizare constă, aşa cum s-a văzut mai sus, din selectarea elementului *x* din mijlocul tabloului, iar cele două părţi se obţin permutînd elementele mai mari sau egale cu *x* din stînga tabloului, cu cele mai mici sau egale cu *x* din dreapta tabloului.

Un element x[i] este în stînga lui x[j], dacă i < j. În mod analog, un element x[k] este în dreapta lui x[j], dacă k > j.

În anumite cazuri este necesar să se permute chiar şi elementul selectat *x*, deoarece în caz contrar nu se poate realiza divizarea. Într-adevăr *x* nu-şi schimbă poziţia dacă în stînga lui sînt atîtea elemente mai mari decît el cîte sînt în dreapta lui mai mici decît el. Evident, această condiţie nu este îndeplinită frecvent şi de aici rezultă necesitatea permutării lui.

Funcţia *quicksort* are parametri:

tab - Tabloul care se sortează.

inf - Indicele inferior al părţii care se divide.

sup - Indicele superior al părţii care se divide.

Dacă *tab* are *n* elemente, atunci funcţia *quicksort* se apelează astfel:

quicksort(tab,0,n-1);

Definim mai jos funcţia *quicksort*.

```
void quicksort(int tab[],int inf,int sup)
/*- sorteaza in ordine crescatoare tabloul tab, folosind metoda sortarii
   rapide;
 - elementul separator este elementul din mijlocul tabloului. */
{
 int i,j,x,t;

 i = inf;
 j = sup;

/* se alege elementul separator */
 x = tab[(i+j)/2];

/* se face divizarea */
 do{
/* avans peste elementele din stinga lui x si mai mici decit el */
      while(i < sup && tab[i] < x)
          i++;

/* avans peste elementele din dreapta lui x mai mari decit el */
      while(j > inf && tab[j] > x)
          j--;
      if(i <= j){
          if(i < j){ /* daca i < j se permuta elementele */
              t = tab[i];
              tab[i] = tab[j];
              tab[j] = t;
          }
```

```
/* se trece la elementele urmatoare */
            i++;
            j--;
        }

/* daca i > j s-a realizat divizarea:
        - prima parte:
                    inf - limita inferioara;
                    j  - limita superioara;
        - partea a doua:
                    i  - limita inferioara;
                    sup - limita superioara;
    daca i <= j, procesul de divizare continua. */

  }while(i <= j);

/* daca inf < j, se trece la divizarea primei parti deoarece aceasta contine
  mai mult decit un element */
  if(inf < j)
      quicksort(tab,inf,j);

/*daca i< sup se trece la divizarea partii a doua deoarece contine mai
  mult decit un element */
  if(i < sup)
      quicksort(tab,i,sup);
}
```

Revenim la problema determinării elementului separator. Pentru detalii, cititorul poate consulta lucrările [3] şi [11].

O metodă relativ simplă şi care dă adesea rezultate bune constă în alegerea *medianei* următoarelor elemente:

- primul element al tabloului;
- elementul aflat la mijlocul tabloului;
- ultimul element al tabloului.

Amintim că dacă *a, b, c* sînt trei numere şi între ele există relaţia

$a < b < c$

atunci *mediana* lor este *b*.

În felul acesta se elimină situaţiile în care elementul separator este elementul minim sau maxim al tabloului.

Propunem cititorului să modifice funcţia *quicksort* în aşa fel încît

elementul separator să fie ales ca mediană a celor trei elemente amintite mai sus.

O altă metodă, descrisă amănunţit în [11] constă în alegerea ca element separator a *primului element al tabloului* şi deplasarea acestuia pe locul lui final în procesul divizării tabloului.

În stînga lui se deplasează elementele mai mici decît el, iar în dreapta cele mai mari decît el. În felul acesta tabloul este divizat în două părţi, iar elementul separator, fiind pe locul lui final, nu aparţine nici uneia din cele două părţi pe care le separă.

Mai jos definim funcţia *quicksort* care utilizează această metodă la alegerea elementului separator.

În principiu, această metodă constă în următoarele:

Notăm cu *inf* şi *sup* (inf < sup) indicii care definesc limita inferioară, respectiv superioară a părţii din tabloul *tab* care urmează a fi divizată.

1. x = tab[inf].
2. i = inf.
3. j = sup.
4. Dacă i < j se trece la pasul 5.

   Altfel la pasul 9.
5. Se determină un element tab[i], incrementînd pe *i* aşa încît tab[i] > x şi i <= sup.
6. Se determină un element tab[j], decrementînd pe *j*, aşa încît tab[j] <= x.
7. Dacă i < j, atunci se permută tab[i] cu tab[j].

   Altfel se trece la pasul 8.
8. Se trece la pasul 4.
9. Se deplasează tab[inf] pe locul lui final permutîndu-l cu tab[j].

   În acest moment tabloul este divizat astfel:

   - prima parte:

     inf - limita inferioară;

     j-1 - limita superioară.
   - tab[j]:

     are poziţia sa finală şi nu se va mai permuta.

– partea a doua:

j+1 - limita inferioară;

sup - limita superioară.

Procesul se reia pentru părţile care nu s-au redus încă la un singur element (limita inferioară a părţii respective este mai mică decît cea superioară).

Exemplificăm metoda pe acelaşi exemplu:

173 25 4 300 256 83 95 14 415 3

inf = 1 şi sup = 10

1. x = tab[1] = 173.
2. i = inf = 1.
3. j = sup = 10.
4. i < j, se trece la pasul 5.
5. tab[i] > 173 pentru i = 4: tab[4] = 300.
6. tab[j] < 173 pentru j = 10: tab[10] = 3.
7. Cum i = 4 < j = 10, se permută cele două elemente:

173 25 4 3 256 83 95 14 415 300.

8. Se reia de la pasul 4.
4. i < j, se trece la pasul 5.
5. tab[5] = 256 > 173, deci i = 5.
6. tab[8] = 14 < 173, deci j = 8.
7. i < j , deci se permută cele 2 elemente:

173 25 4 3 14 83 95 256 415 300.

8. Se reia de la pasul 4.
4. i < j, se trece la pasul 5.
5. tab[8] > 173, deci i = 8.
6. tab[7] < 173, deci j = 7.
7. Cum i > j, nu se face permutare.
8. Se reia de la pasul 4.

4. Cum i > j, se trece la pasul 9.

9. Se permută tab[inf] = tab[1] = 173, cu tab[j] = tab[7] = 95:

   95 25 4 3 14 83 173 256 415 300.

   Tabloul se descompune în:

   - prima parte: A 95 25 4 3 14 83;
   - 173 ocupă locul lui final;
   - partea a doua: B 256 415 300.

   Aplicăm acelaşi algoritm la partea A:

   inf = 1, sup = 6

1. x = tab[inf] = 95.

2. i = 1.

3. j = 6.

4. i < j, se trece la pasul 5.

5. Deoarece tab[i] = 95 este elementul maxim, *i* creşte pînă la valoarea maximă, adică *i* devine egal cu 6.

6. tab[j] = 83 < x = 95, înseamnă că *j* rămîne nemodificat.

7. i = j = 6, deci nu se face permutare.

8. Se reia de la pasul 4.

4. i = j, se trece la pasul 9.

9. Se permută tab[inf] = tab[1] = 95 cu tab[j] = 83:

   83 25 4 3 14 95.

   Tabloul se descompune în:

   - prima parte: AA 83 25 4 3 14;
   - 95 ocupă locul lui final;
   - partea a doua: vidă.

   Aplicăm acelaşi algoritm la partea AA:

   inf = 1, sup = 5

1. x = tab[inf] = 83.

2. i = 1.

3. j = 5.

4. i < j, se trece la pasul 5.

5. Deoarece tab[i] = 83 este elementul maxim, *i* creşte pînă la valoarea maximă, adică *i* devine egal cu 5.

6. tab[j] = 14 < x = 83, înseamnă că *j* rămîne nemodificat.

7. i = j = 5, deci nu se face permutare.

8. Se reia de la pasul 4.

4. i = j, se trece la pasul 9.

9. Se permută tab[inf] = tab[1] = 83 cu tab[j] = 14:

   14 25 4 3 83.

   Tabloul se descompune în:

   - prima parte: AAA 14 25 4 3;
   - 83 ocupă locul lui final;
   - partea a doua: vidă.

   Aplicăm acelaşi algoritm la partea AAA:

   inf = 1, sup = 4

1. x = tab[inf] = tab[1] = 14.

2. i = 1.

3. j = 4.

4. i < j, se trece la pasul 5.

5. *i* creşte la valoarea 2, deoarece tab[2] = 25 > x.

6. *j* nu se modifică deoarece tab[j] = tab[4] = 3 < x.

7. i < j, se permută tab[2] cu tab[4]:

   14 3 4 25.

8. Se reia de la pasul 4.

4. i < j, se trece la pasul 5.

5. *i* creşte pînă la 4, deoarece tab[4] = 25 > x.

6. *j* devine egal cu 3, deoarece tab[3] = 4 < x.

7. i > j, deci nu se face permutare.

8. Se trece la pasul 4.

4. i > j, deci se trece la pasul 9.

9. Se permută tab[inf] = tab[1] = 14 cu tab[j] = tab[3] = 4:

   4 3 14 25.

   Tabloul se descompune în:

   - prima parte: AAAA 4 3;
   - 14 ocupă locul lui final;
   - partea a doua: 25.

   Aplicăm acelaşi algoritm la partea AAAA:

   inf = 1, sup = 2

1. x = tab[inf] = 4.

2. i = 1.

3. j = 2.

4. i < j, se trece la pasul 5.

5. Deoarece tab[i] = 4 este elementul maxim, *i* creşte pînă la valoarea maximă, adică *i* devine egal cu 2.

6. tab[j] =tab[2]= 3 < x = 4.

   Înseamnă că *j* rămîne nemodificat.

7. i = j = 2, deci nu se face permutare.

8. Se reia de la pasul 4.

4. i = j, se trece la pasul 9.

9. Se permută tab[inf] cu tab[j]:

   3 4.

   Tabloul se descompune în:

   - prima parte: 3;
   - 4 ocupă locul lui final;
   - partea a doua: vidă.

   În acest moment elementele tabloului sînt ordonate astfel:

   3 4 14 25 83 95 173 256 415 300.

Ultimele 3 elemente formează partea B de la prima diviziune a aceleeaşi metode:

inf = 8, sup = 10

1. x = tab[inf] = tab[8] = 256.
2. i = 8.
3. j = 10.
4. i < j, se trece la pasul 5.
5. i devine egal cu 9 deoarece tab[9] = 415 > x.
6. j devine egal cu 8 deoarece tab[10] > x şi tab[9]>x, tab[8] fiind primul care nu are această proprietate.
7. i > j, deci nu se face permutare.
8. Se reia de la pasul 4.
4. i > j, se trece la pasul 9.
9. Cum inf = j, nu se face permutare.

Tabloul se descompune în:

- prima parte: vidă;
- 256 ocupă locul lui final;
- partea a doua: BB 415 300;

Aplicăm acelaşi algoritm la partea BB şi în final se obţine:

3 4 14 25 83 95 173 256 300 415.

Mai jos, transcriem acest procedeu în limbajul C.

```
void quicksortinf(int tab[],int inf,int sup)
/* - sorteaza in ordine crescatoare tabloul tab folosind metoda sortarii
     rapide;
 - elementul separator este tab[inf] care se deplasează pe locul lui
     final. */
{
 int i,j,t,x;

 x = tab[inf];
 i = inf;
 j = sup;

/* divizarea tabloului */
```

```
while(i < j){

/*- se determina cel mai mic indice i astfel ca tab[i] > x şi i <= sup;
  - se avanseaza peste elementele mai mici sau egale cu x;
  - i = sup daca nu exista un astfel de element. */
    while(tab[i] <= x && i < sup)
        i++;

/* - se determina cel mai mare indice j astfel ca tab[j]<=x;
   - se avanseaza peste elementele mai mari decit x. */
    while(tab[j] > x)
        j--;

/* daca i < j, se permuta tab[i] cu tab[j] */
    if(i < j){ /* se face permutare */
        t = tab[i]
        tab[i] = tab[j];
        tab[j] = t;
    }
}

/* - s-a terminat divizarea tabloului;
   - tab[inf] se deplaseaza pe locul lui final care este j;
   - totodata tab[j] poate fi deplasat in pozitia inf;
   - deci se permuta tab[inf] cu tab[j];
   - permutarea nu are loc daca j = inf. */
if(j > inf){
    tab[inf] = tab[j];
    tab[j] = x;
}

/* s-a realizat divizarea:
     - prima parte:
             tab[inf],tab[inf+1],...,tab[j-1];
     - tab[j] ocupa pozitia lui finala;
     - partea doua:
             tab[j+1],tab[j+2],...,tab[sup]. */
if(inf < j-1)

/* prima parte are cel putin 2 elemente */
    quicksortinf(tab,inf,j-1);

if(j+1 < sup)
```

```
/* partea a doua are cel putin 2 elemente */
    quicksortinf(tab,j+1,sup);
}
```

Menţionăm că în lucrarea [11], această metodă este transcrisă în limbajul C într-o funcţie nerecursivă.

Exerciţii:

14.1 Să se scrie un program care realizează următoarele:

- citeşte o succesiune de numere separate prin caractere albe şi precedate de numărul lor;
- sortează în ordine crescătoare numerele citite şi apoi le afişează; sortarea se realizează folosind următoarele metode:
  - metoda bulelor;
  - sortare Shell;
  - sortare rapidă cu alegerea elementului din mijloc ca element separator;
  - sortare rapidă cu alegerea primului element ca element separator.
- afişează numărul permutărilor efectuate în cadrul fiecărei metode de sortare utilizate.

PROGRAMUL BXIV1

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

#include "BVIII2.CPP" /* pcit_int     */
#include "BVIII3.CPP" /* pcit_int_lim */
#include "BVIII8.CPP" /* pndcit       */
#include "BVIII9.CPP" /* pdvcit       */

#define MAX 100

void dperm(double *p1,double *p2)
/* permuta valorile de tip double spre care pointeaza p1 si p2 */
{
 double temp;
```

```
 temp = *p1;
 *p1 = *p2;
 *p2 = temp;
}

void dcopiaza(double tab1[],double tab2[],int n)
/* copiaza primele n elemente din tabloul tab2 in tabloul tab1 */
{
 int i;

 for(i=0;i < n;i++)
     tab1[i]=tab2[i];
}

void afistab(double tab[],int n)
/* - afiseaza primele n elemente din tabloul tab;
   - pe o linie se afiseaza 5 elemente. */
{
 char text[] = "Actionati o tasta pentru a\
                continua\n";
 int i,j;
 double d;

 printf("\n");
 j=1;
 for(i = 0;i < n;i++){
     d = tab[i];
     printf("%f ",d);
     if(i%5==4){
         printf("\n");
         if(j++ %22==0){
             printf(text);
             getch();
         }
     }
 }
 printf(text);
 getch();
}

void afisantet(char *p) /* afiseaza un antet */
{
 printf("\n\t\t%s\n",p);
```

```
 printf("Actionati o tasta pentru a continua\n");
 getch();
}

void afisnrperm(char *p,long n)
/* afiseaza textul spre care pointeaza p si valoarea lui n */
{
 printf("\n Metoda %s\t numar permutari=%ld\n",
        p,n);
}

long bubblesort(double tab[],int n)
/* - sorteaza elementele tabloului tab in ordine crescatoare folosind
     metoda bulelor;
   - returneaza numarul permutarilor efectuate. */
{
 int ind,i;
 long s;

 ind = 1;
 s=0;
 while(ind){
     ind = 0;
     for(i = 0;i < n-1;i++)
     if(tab[i] > tab[i+1]){
         dperm(&tab[i],&tab[i+1]);
         s++; /* numara permutari */
         ind = 1;
     } /* sfirsit if */
 } /* sfirsit while */
 return s;
} /* sfirsit bubblesort */

long shellsort(double tab[],int n)
/* - sorteaza elementele tabloului tab in ordine crescatoare folosind
     metoda lui Shell;
   - returneaza numarul permutarilor efectuate. */
{
 int inc;
 int i,j;
 long s;

 s = 0;
```

```
 for(inc = n/2;inc > 0;inc /= 2)
     for(i = inc; i < n;i++)
         for(j = i-inc;j >= 0 &&
               tab[j] > tab[j+inc];j -= inc){
             dperm(&tab[j],&tab[j+inc]);
             s++;
         }
 return s;
} /* sfirsit shellsort */

void quicksort(double tab[],int inf,int sup)
/*- sorteaza elementele tabloului tab in ordine crescatoare folosind
    metoda sortarii rapide;
  - elementul separator este elementul din mijlocul tabloului;
  - numarul permutarilor realizate se atribuie variabilei globale s. */
{
 extern long s;
 int i,j;
 double x;

 i = inf;
 j = sup;
 x = tab[(i+j)/2];
 do{
     while(i < sup && tab[i] < x)
         i++;
     while(j > inf && tab[j] > x)
         j--;
     if(i <= j){
         if(i < j){
             dperm(&tab[i],&tab[j]);
             s++;
         }
         i++;
         j--;
     }
 }while(i <= j);
 if(inf < j)
     quicksort(tab,inf,j);
 if(i < sup)
     quicksort(tab,i,sup);
} /* sfirsit quicksort */
```

```
void quicksortinf(double tab[],int inf,int sup)
/* - sorteaza elementele tabloului tab in ordine crescatoare folosind
     metoda sortarii rapide;
   - elementul separator este primul element al tabloului si acesta se
     deplaseaza pe locul lui final;
   - numarul permutarilor realizate se atribuie variabilei globale s. */
{
 extern long s;
 int i,j;
 double x;

 x = tab[inf];
 i = inf;
 j = sup;
 while(i < j){
     while(tab[i] <= x && i < sup)
         i++;
     while(tab[j] > x)
         j--;
     if(i < j){
         dperm(&tab[i],&tab[j]);
         s++;
     }
 }
 if(j > inf){
     tab[inf] = tab[j];
     tab[j] = x;
     s++;
 }
 if(inf < j-1)
     quicksortinf(tab,inf,j-1);
 if(j+1 < sup)
     quicksortinf(tab,j+1,sup);
} /* sfirsit quicksortinf */

long s;

main()
/* - citeste o succesiune de numere separate prin caractere albe si
     precedate de numarul lor;
   - sorteaza in ordine crescatoare numerele citite si apoi le afiseaza;
   - sortarea se face folosind urmatoarele metode:
         metoda bulelor;
```

```
        sortare shell;
        sortare rapida cu alegerea elementului din mijloc ca separator
        sortare rapida cu alegerea primului element ca element separator
   - afiseaza numarul permutarilor efectuate in cadrul fiecarei metode
     utilizate. */
{
 double tini[MAX],tsort[MAX];
 int n;
 long a,b,c,d;

/* citeste numerele de sortat */
 if((n = pdvcit(MAX,tini)) < 2){
     printf("nu s-au tastat nici 2 numere\n");
     exit(1);
 }

/* sortare prin metoda bulelor */
 dcopiaza(tsort,tini,n); /* tsort = tini */
 a = bubblesort(tsort,n);
 afisantet("metoda bulelor");
 afistab(tsort,n);

/* sortare prin metoda Shell */
 dcopiaza(tsort,tini,n);
 b = shellsort(tsort,n);
 afisantet("metoda Sell");
 afistab(tsort,n);

/* sortare rapida - elementul separator este elementul din mijlocul
                    tabloului */
 dcopiaza(tsort,tini,n);
 s = 0;
 quicksort(tsort,0,n-1);
 c = s;
 afisantet("sortare rapida: elementul din mijloc");
 afistab(tsort,n);

/* sortare rapida - elementul separator este primul element */
 dcopiaza(tsort,tini,n);
 s = 0;
 quicksortinf(tsort,0,n-1);
 d = s;
 afisantet("sortare rapida: primul element");
```

```
 afistab(tsort,n);

/* se afiseaza numarul permutarilor */
 afisnrperm("Bulelor",a);
 afisnrperm("Shell",b);
 afisnrperm("Sortare rapida: mijloc",c);
 afisnrperm("Sortare rapida: inferior",d);
}
```

15. DIN NOU DESPRE PREPROCESARE ÎN C

În capitolul 1 s-a arătat că preprocesarea este o fază care precede compilarea propriu-zisă. Preprocesorul limbajului C este relativ simplu. El, în principiu, execută *substituţii* de texte. Prin intermediul lui se pot realiza:

- includeri de fişiere sursă;
- definiţii şi apeluri de macrouri;
- compilare condiţionată.

Preprocesorul recunoaşte construcţiile care încep prin caracterul diez (#). Acestea se mai numesc şi *directive*.

Aşa de exemplu, includerile de fişiere se realizează cu ajutorul construcţiei *#include*. Această construcţie a fost descrisă în capitolul 1.

În capitolul de faţă vom reveni asupra construcţiei *#define*.

În general, construcţia *#define* se foloseşte la definirea de macrouri.

Apoi, ne vom ocupa de compilarea condiţionată.

15.1. Definiţii şi apeluri de macrouri

Definiţia constantelor simbolice este un caz particular de definiţie de macro.

Amintim că o constantă simbolică se defineşte printr-o construcţie de forma:

#define *nume succesiune de caractere*

Efectul acestei construcţii constă în substituirea în textul programului a numelui aflat după #define cu *succesiune de caractere* din construcţia *#define* respectivă.

Substituţia se realizează pentru orice apariţie a lui *nume* în textul sursă care urmează după definirea lui prin construcţia *#define*, exceptînd cazurile în care *nume* apare într-un şir de caractere sau într-un comentariu.

Efectul construcţiei *#define* se anulează la întîlnirea construcţiei:

#undef *nume*

După o astfel de construcţie, *nume* poate fi redefinit printr-o altă

construcţie *#define*.

La scrierea construcţiei *#define*, *nume* este precedat şi urmat de cel puţin un spaţiu. Succesiunea de caractere care se substituie numelui începe cu primul caracter care nu este alb. Ea se poate continua pe rîndul următor terminînd-o prin caracterul backslash (\).

Un *macro* este o facilitate generală de prelucrare a textelor prin substituţie. Un macro, ca şi o funcţie, are o definiţie şi unul sau mai multe apeluri. Definiţia macroului trebuie să preceadă orice apel al său. Ea defineşte textul care urmează a se substitui la fiecare apel al macroului. Acest text poate fi variabil, variind de la apel la apel. De aceea, un astfel de text, de obicei, va conţine parametri. Aceştia se concretizează la fiecare apel. Parametrii utilizaţi în definiţia macroului se numesc *formali*. Valorile lor dintr-un apel al macroului sînt parametrii *efectivi* sau *concreţi*.

Forma generală a unei definiţii de macro este următoarea:

#define *nume(p1,p2,...,pn) text*

unde:

nume	- Este numele macroului.
p1,p2,...,pn	- Sînt parametri formali ai macroului.
text	- Este textul de substituţie.
	- El conţine parametri formali p1,p2,...,pn.
	- Dacă este nevoie, el se poate continua pe rîndul următor folosind caracterul *backslash*.

Fiecare p1,p2,...,pn este un nume.

Dacă macroul nu are parametri, atunci este absentă toată construcţia (p1,p2,...,pn) şi evident, textul de substituţie devine constant (nu mai conţine parametri). În acest caz definiţia de macro devine o definiţie de constantă simbolică.

În scrierea unei definiţii de macro cu parametri trebuie ca între numele macroului şi paranteza deschisă să *nu* existe nici un caracter alb. În caz contrar, macroul se consideră fără parametri şi paranteza închisă, ce conţine lista parametrilor, se consideră ca făcînd parte din textul de substituţie.

Apelul unui macro constă din numele lui urmat de lista valorilor parametrilor inclusă între paranteze rotunde.

Parametri de la apel (efectivi) se separă prin virgulă dacă sînt mai mulţi. Ei se substituie parametrilor formali în textul de substituţie al macroului apelat şi apoi textul rezultat se substituie apelului.

Corespondenţa dintre parametri formali şi valorile lor de la apel se realizează prin poziţie.

Substituirea apelului unui macro prin textul de substituţie după ce, în prealabil, eventualii parametri formali au fost înlocuiţi cu valorile lor din apel, se numeşte *expandare*.

Expandarea poate fi anihilată cu ajutorul construcţiei *#undef*:

#undef *nume*

unde:

nume - Este numele macroului.

O definiţie de macro este valabilă din punctul în care este scrisă şi pînă la sfîrşitul textului sursă în care apare sau pînă la întîlnirea construcţiei *#undef* care o anulează.

Apelul unui macro se expandează dacă acesta se află în textul sursă în zona în care este valabilă definiţia lui. Evident, un apel de macro nu se expandează dacă se află într-un comentariu sau într-un şir de caractere.

Exemple:

1. Macroul de mai jos defineşte maximul dintre valorile a două expresii. În acest scop se foloseşte o expresie condiţională.

   Macroul se defineşte în felul următor:

   ```
   #define MAX(x,y) ((x) > (y) ? (x) : (y))
   ```

Dăm mai jos exemple de apeluri ale acestui macro:

a.

```
int i,j,k;
...
k = MAX(i+j,i-j);
```

Apelul MAX(i+j,i-j) se expandează la preprocesare, iar la compilare se întîlneşte instrucţiunea de atribuire de mai jos:

```
k = ((i+j) > (i-j) ? (i+j) : (i-j));
```

b.

```
float a,b,c;
...
c = MAX(a,b);
```

Rezultatul expandării este:

```
c = ((a) > (b) ? (a) : (b));
```

2. Mai jos, se defineşte un macro pentru calculul valorii absolute:

```
#define ABS(x) ((x) < 0 ? -(x) : (x))
```

Exemple de apeluri şi expandări ale acestui macro:

a.

```
int i,j,k;
...
k = ABS(i-j);
```

Se expandează astfel:

```
k = ((i-j) < 0 ? -(i-j) : (i-j));
```

b.

```
double a,b,c,eps;
...
c = ABS(a-b) < eps;
```

Se expandează astfel:

```
c = ((a-b) < 0 ? -(a-b) : (a-b)) < eps;
```

3. Macrourile de mai jos se utilizează la transformarea literelor mici în litere mari şi invers:

```
#define UPPER(c) ((c)-'a'+'A')
#define LOWER(c) ((c)-'A'+'a')
```

Exemple de apel:

putchar(UPPER(c)); scrie o literă mare dacă valoarea lui *c* este codul ASCII al unei litere mici;

putchar(LOWER(c)); scrie o literă mică dacă valoarea lui *c* este codul ASCII al unei litere mari.

4. Macroul de mai jos permite permutarea a două valori de tip *int*:

```
#define IPERM(X,Y) {\
 int t;\
 t = X;\
 X = Y;\
 Y = t;\
}
```

Apelul:

IPERM (a,b)

se expandează astfel:

```
{
 int t;

 t = a;
 a = b;
 b = t;
}
```

Putem realiza un macro mai general de permutare care să fie valabil pentru date numerice de orice tip predefinit:

```
#define PERM(TIP,X,Y) {\
TIP t;\
t = X;\
X = Y;\
Y = t;\
}
```

Apelul

PERM(int,a,b)

se expandează la fel ca şi IPERM.

Pentru a permuta date flotante vom folosi apelul:

PERM(float,a,b)

Se observă o analogie între funcţii şi macrouri. Ambele au o definiţie şi unul sau mai multe apeluri. De asemenea, atît în definiţia de funcţie cît şi în cea de macro se pot utiliza parametri formali. În ambele cazuri, valorile parametrilor formali se definesc la apeluri. Există însă o diferenţă esenţială între cele două moduri de definire a valorilor parametrilor.

În cazul macrourilor, valoarea parametrului formal este succesiunea de caractere corespunzătoare lui din apel. Această succesiune de caractere se substituie în locul parametrului formal peste tot unde acesta apare în textul de substituţie al macroului. De exemplu, la apelul PERM(int,a,b) de mai sus, parametrul formal TIP se substituie prin succesiunea de caractere *int* în textul de substituţie şi în felul acesta se obţine declaraţia:

int t;

În cazul funcţiilor, parametrilor formali li se atribuie valorile parametrilor de la apel care sînt date de tipuri predefinite sau adrese de zone de memorie.

O altă diferenţă esenţială constă în aceea că, în timp ce apelul unei funcţii înseamnă un salt la o zonă de memorie în care se păstrează instrucţiunile în format executabil corespunzător funcţiei respective, în cazul macrourilor apelul unui macro se înlocuieşte chiar cu instrucţiunile rezultate în urma compilării, expandării macroului respectiv.

De exemplu, fie funcţia:

```
int abs(int x) /* returneaza valoarea absoluta a lui x */
{
return x < 0 ? -x : x;
}
```

şi apelul ei:

```
int i,j;
...
j = abs(i);
...
```

La apelul funcţiei se realizează un salt la zona în care se află rezultatul compilării funcţiei *abs* definită mai sus.

După executarea instrucţiunilor corespunzătoare funcţiei *abs*, se revine în punctul de apel şi se atribuie lui *j* valoarea returnată de funcţie (valoarea absolută a lui *i*).

Înlocuind definiţia funcţiei *abs* prin definiţia de macro de mai jos:

```
#define abs(x) ((x) < 0 ? -(x) : (x))
```

acelaşi apel se expandează astfel:

```
j = ((i) < 0 ? -(i) : (i));
```

În acest caz, apelul macroului se înlocuieşte prin instrucţiunile care calculează valoarea absolută a lui *i*.

Cu alte cuvinte, în cazul macrourilor instrucţiunile definite de ele se generează şi apoi se compilează direct în locurile apelurilor, adică în poziţia în care este nevoie de ele. În cazul funcţiilor, instrucţiunile definite de corpul unei funcţii se păstrează într-o zonă de memorie a cărei poziţie nu are nimic comun cu apelurile funcţiei. La fiecare apel al funcţiei se realizează un salt la zona respectivă, iar după terminarea execuţiei funcţiei se revine la punctul de după apel.

Avînd în vedere cele de mai sus, se pune problema existenţei avantajelor la utilizarea macrourilor în locul funcţiilor.

Utilizarea macrourilor este avantajoasă pentru procese de calcul foarte

simple, cum sînt exemplele de mai sus: maximul dintre două valori numerice, valoarea absolută, permutarea valorilor a două variabile etc.

Într-adevăr, prin utilizarea macrourilor, în locul funcţiilor, se elimină apelurile funcţiilor. Amintim că la apelul unei funcţii se face nu numai salt la zona de memorie corespunzătoare funcţiei apelate, ci se alocă pe stivă parametri formali şi variabilele automatice ale funcţiei respective. De asemenea, la revenirea din funcţie se face curăţirea stivei. Toate aceste activităţi pot fi mai complexe decît însuşi procesul de calcul realizat prin funcţie. De aceea, în acest caz, se preferă utilizarea macrourilor în locul funcţiilor.

Prin utilizarea macrourilor nu se mai face apel şi nici alocări de parametri pe stivă şi respectiv dezalocarea lor. Apelul macroului pur şi simplu se înlocuieşte prin instrucţiunile care realizează procesul de calcul exprimat prin expandarea macroului. În acest caz se obişnuieşte să se spună că macroul defineşte o funcţie generată *în linie* (in-line).

Pentru procese de calcul mai complexe nu se justifică utilizarea macrourilor, deoarece substituirea fiecărui apel prin textul de substituţie al macroului apelat, va conduce la un consum mare de memorie.

În limbajul C++ se pot defini funcţii *in-line*. Acestea, ca şi macrourile, se generează pe locul apelurilor lor. De aceea, o funcţie *in-line* trebuie să corespundă unui calcul simplu care se poate realiza cu un număr mic de instrucţiuni (2-3 instrucţiuni). În caz contrar, se poate ajunge la un consum mare de memorie.

Diferenţa dintre funcţiile *in-line* şi macrouri constă în faptul că, la apelul funcţiilor in-line se fac controale asupra tipului parametrilor utilizaţi la apel, în timp ce în cazul macrourilor astfel de controale nu mai au loc.

15.2. Compilare condiţionată

Compilarea condiţionată permite să se aleagă,dintr-un text general, părţile care să se compileze împreună. Acest lucru este util pentru programe care au părţi comune sau care depind de calculator sau de versiunea compilatorului. În felul acesta, în funcţie de anumiţi parametri, se pot alege părţile din textul sursă care urmează a fi compilate împreună.

Compilarea condiţionată se realizează folosind construcţiile (directivele) preprocesorului:

#if, **#ifdef** şi **#ifndef**.

Construcţia *#if* are două formate:

#if *expr*
text
#endif

şi

#if *expr*
text1
#else
text2
#endif

unde:

expr - Este o expresie constantă (valoarea ei poate fi evaluată de preprocesor la întîlnirea ei).

text, text1 şi text2 - Sînt texte sursă care sînt supuse compilării în funcţie de valoarea expresiei *expr*.

În primul format, dacă *expr* are valoarea adevărat (este diferită de zero), atunci *text* se supune preprocesării şi în consecinţă şi compilării, apoi se continuă cu textul sursă aflat după *#endif*. În caz contrar (*expr* are valoarea zero), se continuă cu textul aflat după *#endif*.

În formatul al doilea, dacă *expr* are valoarea adevărat, atunci se supune preprocesării *text1* şi deci şi compilării apoi se continuă cu textul care urmează după *#endif*. În caz contrar (*expr* are valoarea zero) se supune preprocesării *text2* şi deci şi compilării, apoi se continuă cu textul aflat după *#endif*.

Directivele *#ifdef* şi *#ifndef* se utilizează în mod analog.

În cazul directivei *#ifdef* se utilizează următoarele formate:

#ifdef *nume*
text
#endif

şi

#ifdef *nume*
text1
#else
text2
#endif

În primul format, *text* se supune preprocesării şi deci şi compilării, dacă *nume* este definit în momentul întîlnirii directivei *#ifdef* de către preprocesor.

În formatul al doilea *text1* se preprocesează şi apoi compilează numai

dacă *nume* este definit în momentul întîlnirii directivei *#ifdef* de către preprocesor. În caz contrar (*nume* nu este în prealabil definit) se preprocesează şi apoi compilează *text2*.

Directiva *#ifndef* are o acţiune opusă directivei *#ifdef*. Deci, *text* din formatul:

#ifndef *nume*
    text
#endif

se preprocesează şi apoi compilează dacă *nume* nu este definit în momentul întîlnirii directivei *#ifndef* de către preprocesor.

În mod analog, formatul:

#ifndef *nume*
    text1
#else
    text2
#endif

realizează preprocesarea şi apoi compilarea lui *text1* dacă *nume* nu este definit în momentul întîlnirii directivei *#ifndef* de către preprocesor. În caz contrar (*nume* este în prealabil definit) se preprocesează şi apoi compilează *text2*.

Exemplu:

Un program se compune din două fişiere *fis1.cpp* şi *fis2.cpp*. Ambele fişiere conţin apeluri la funcţii care au prototipurile în fişierul *stdio.h*.

Întrucît cele două fişiere au fost editate de programatori diferiţi, ele au fost la început compilate separat pentru a se elimina erorile sintactice. De aceea, fiecare fişier conţine la începutul lui directiva:

```
#include <stdio.h>
```

În final, cele două fişiere se compilează împreună inserînd directiva:

```
#include "fis1.cpp"
```

în fişierul fis2.cpp sau directiva:

```
#include "fis2.cpp"
```

în fişierul fis1.cpp.

În acest caz fişierul *stdio.h* se va include (preprocesa) de două ori. Pentru a evita acest lucru se pot folosi directivele compilării condiţionate, ca mai jos.

Se inserează la începutul fiecărui fişier secvenţa:

```
#ifndef __STDIO_H
#include <stdio.h>
#define __STDIO_H
#endif
```

Această secvenţă de directive realizează următoarele:

- dacă numele __STDIO_H nu este definit, atunci:
  - se include fişierul *stdio.h*;
  - se defineşte numele __STDIO_H.
- în caz contrar (numele __STDIO_H este definit) se trece la preprocesarea textului aflat după *#endif*.

Se observă că această secvenţă permite includerea fişierului *stdio.h* numai în cazul în care __STDIO_H este nedefinit. Totodată, după includerea lui *stdio.h* se defineşte variabila __STDIO_H, ceea ce permite suprimarea includerilor ulterioare ale fişierului *stdio.h*.

Directiva *#define* de mai sus nu atribuie o valoare de substituţie pentru numele __STDIO_H deoarece acest nume nu este folosit la substituţie ci numai în directiva *#ifndef*.

Numele __STDIO_H a fost ales ca să fie sugestiv şi pe cît posibil să nu coincidă cu unul care să aibă alte utilizări.

Exerciţii:

15.1 Să se definească un macro care apoi să fie apelat la evaluarea expresiilor de mai jos:

$$i = (|a| + |b|) / (1 + |a-b|)$$

unde:

i, a, b - Sînt de tip *int*.

$$z = (|x+y| - |x-y|) / (1 + |x+y|\,|x-y|)$$

unde:

x, y, z - Sînt de tip *double*.

Definiţia macroului şi apelurile lui pentru evaluarea acestor expresii intră în compunerea programului de mai jos.

Valorile variabilelor a, b, x şi y se citesc de la intrarea standard.

PROGRAMUL BXV1

```
#include <stdio.h>
#include <stdlib.h>

#define ABS(X) ((X)<0 ? -(X) : (X))

#include "BVIII2.CPP" /* pcit_int     */
#include "BVIII3.CPP" /* pcit_int_lim */

int pcit_double(char *p,double *d);

main() /* citeste valorile variabilelor a, b, x si y, evalueaza expresiile de
          mai jos si afiseaza valorile variabilelor i si z:
                i=(|a|+|b|)/(1+|a-b|);
                z=(|x+y|-|x-y|)/(1+|x+y||x-y|). */
{
 int a,b,i;
 double x,y,z;
 double s,d;
 char er[] = "s-a tastat EOF\n";

/* citeste valoarea lui a */
 if(pcit_int_lim("a=",-32768,32767,&a)==0){
     printf(er);
     exit(1);
 }

/* citeste valoarea lui b */
 if(pcit_int_lim("b=",-32768,32767,&b)==0){
     printf(er);
     exit(1);
 }
/* citeste valoarea lui x */
 if(pcit_double("x=",&x)==0){
     printf(er);
     exit(1);
 }

/* citeste valoarea lui y */
 if(pcit_double("y=",&y)==0){
     printf(er);
     exit(1);
 }
```

```
/* calculul valorii lui i */
 i = (ABS(a)+ABS(b)) / (1+ABS(a-b));

/* calculul valorii lui z */
 s = ABS(x+y);
 d = ABS(x-y);
 z = (s-d) / (1+s*d);

/* afisare rezultate */
 printf("a = %d\tb = %d\n",a,b);
 printf("(ABS(a)+ABS(b))/(1+ABS(a-b))=%d\n",i);
 printf("x = %f\ty = %f\n",x,y);
 printf("(ABS(x+y)-ABS(x-y))/(1+ABS(x+y)*\
       ABS(x-y))=%g\n",z);

} /* sfirsit main */

int pcit_double(char *p,double *d)
/* - afiseaza textul spre care pointeaza p;
  - citeste un numar si-l pastreaza in zona spre care pointeaza d;
  - returneaza zero la intilnirea sfirsitului de fisier si unu in caz
    contrar. */
{
 char t[255];
 double f;

 for(;;){
     printf("%s",p);
     if(gets(t)==0)
         return 0;
     if(sscanf(t,"%lf",&f)==1)
         break;
     printf("nu s-a tastat un numar\n");
 }
 *d=f;
 return 1;
}
```

15.2 Să se scrie un program în care să se definească un macro pentru calculul unui polinom de gradul doi şi care să fie apelat pentru evaluarea expresiilor:

poll = 3x*x+7x-8;

pol2 = x*x-3,5x+1,2.

Programul citeşte valoarea lui x şi afişează valorile variabilelor *pol1* şi *pol2*.

PROGRAMUL BXV2

```
#include <stdio.h>
#include <stdlib.h>

#define POL(A,B,C) ((A)*x*x+(B)*x+(C))

main() /* citeste pe x, calculeaza si afiseaza valorile expresiilor:
            3x*x+7x-8
          si
            x*x-3,5x+1,2. */
{
 double x;
 char t[255];

 for(;;){
     printf("x=");
     if(gets(t)==0){
         printf("s-a tastat EOF\n");
         exit(1);
     }
     if(sscanf(t,"%lf",&x)==1)
         break;
     printf("nu s-a tastat un numar\n");
 }
 printf("x = %f\t 3x*x+7x-8 = %g\n",x,POL(3,7,-8));
 printf("x = %f\t x*x-3,5x+1,2 = %g\n",x,
        POL(1,-3.5,1.2));
}
```

16. INTRĂRI/IEŞIRI

Limbajul C nu dispune de instrucţiuni de intrare/ieşire. Aceste operaţii se realizează prin intermediul unor *funcţii* din biblioteca standard a limbajului . Aceste funcţii pot fi aplicate în mod eficient la o gamă largă de aplicaţii. De asemenea, ele asigură o portabilitate bună a programelor, fiind implementate într-o formă compatibilă pe diferite sisteme de operare. Aceasta însă nu înseamnă că ele nu au şi facilităţi suplimentare pe anumite sisteme, cum ar fi de exemplu în cazul mediului de programare Turbo C.

În acest capitol ne vom referi la funcţiile din biblioteca standard I/O care au o utilizare frecventă în diferite aplicaţii. Deşi ele au un caracter general, în capitolul de faţă ne vom referi la facilităţile oferite de funcţiile existente în biblioteca compilatorului Turbo C.

Majoritatea operaţiilor de intrare/ieşire se realizează în ipoteza că datele sînt organizate în *fişiere*.

În general, prin *fişier* înţelegem o colecţie ordonată de elemente,numite *înregistrări*, care sînt păstrate pe diferite suporturi externe. Cele mai utilizate suporturi pentru fişiere sînt cele magnetice. Acestea, de obicei, sînt *discuri* şi *benzi magnetice*. Ele se numesc suporturi *reutilizabile*, deoarece zona utilizată pentru a păstra înregistrările unui fişier poate fi ulterior reutilizată pentru a păstra înregistrările altui fişier.

Datele introduse de la tastatura unui terminal se consideră că formează un *fişier de intrare*. Înregistrarea în acest caz, de obicei, este formată din datele tastate la terminal pe un rînd deci caracterul de rînd nou (*newline*) este *terminator de înregistrare*. În mod analog, datele care se afişează pe terminal formează un *fişier de ieşire*. Înregistrarea, şi în acest caz, poate fi formată din caracterele unui rînd.

Datele care se scriu la imprimantă formează şi ele un fişier de ieşire. Un rînd scris la imprimantă este o înregistrare.

Un fişier are o înregistrare care marchează *sfîrşitul de fişier*. În cazul fişierelor de intrare de la tastatură sfîrşitul de fişier se generează prin secvenţa:

<Ctrl>-Z

De obicei, prin *intrare standard* se înţelege tastatura terminalului de la care s-a lansat programul. În mod analog; *ieşirea standard* este ecranul aceluiaşi terminal.

Prelucrarea fişierelor implică un număr de *operaţii* specifice acestora. În

primul rînd, orice fişier, înainte de a fi prelucrat, trebuie *deschis*. De asemenea, la terminarea prelucrării unui fişier, acesta trebuie *închis*.

Alte operaţii frecvente în prelucrarea fişierelor sînt:

- crearea unui fişier;
- citirea (consultarea) înregistrărilor unui fişier;
- actualizarea unui fişier;
- adăugarea de înregistrări într-un fişier;
- poziţionarea într-un fişier;
- ştergerea unui fişier.

Toate aceste operaţii pot fi realizate prin funcţii standard existente în biblioteca limbajului C.

Tratarea fişierelor se poate face la două *nivele*. *Primul nivel* face apel direct la sistemul de operare. Acesta este *nivelul inferior* de prelucrare a fişierelor.

Celălalt nivel se realizează prin intermediul unor proceduri specializate de prelucrare a fişierelor care utilizează structuri speciale de tip FILE (fişier). Acesta este *nivelul superior* de prelucrare a fişierelor.

Pînă în prezent am utilizat funcţii pentru prelucrarea la nivel superior a fişierelor standard de intrări/ieşiri la terminalul de la care s-a lansat programul.

Aşa de exemplu, pentru a citi o înregistrare de la intrarea standard am utilizat funcţia *gets*. Pentru a afişa înregistrarea la terminalul standard am folosit funcţia *puts*. Alte funcţii utilizate, care permit şi conversii din format intern în cel extern şi invers, sînt funcţiile *scanf* şi *printf*. Prima permite citirea datelor de la intrarea standard sub controlul formatelor. Cea de a doua funcţie permite afişarea standard sub controlul formatelor.

Aceste funcţii au prototipurile în fişierul *stdio.h*.

Pentru citirea de caractere de la intrarea standard am folosit macroul *getchar* definit în fişierul *stdio.h*. În mod analog, pentru a afişa caractere la ieşirea standard am folosit macroul *putchar* definit tot în fişierul *stdio.h*. În continuare se indică funcţii standard analoge din biblioteca C care pot fi utilizate pentru prelucrarea altor fişiere decît cele de la intrarea sau ieşirea standard.

16.1. Nivelul inferior de prelucrare a fişierelor

16.1.1. Deschiderea unui fişier

Aşa cum s-a spus mai sus, orice fişier înainte de a fi prelucrat trebuie *deschis*. Deschiderea unui fişier existent se realizează cu ajutorul funcţiei *open*. La revenirea din funcţia *open* se returnează aşa numitul *descriptor de fişier*. Acesta este număr întreg nenegativ. El identifică în continuare fişierul respectiv în toate operaţiile realizate asupra lui.

În forma cea mai simplă, funcţia *open* se apelează printr-o expresie de atribuire de forma:

df = **open**(...);

unde:

df - Este o variabilă de tip *int*.

Funcţia *open* are prototipul:

int open(const char **cale*, **int** *acces*);

unde:

cale - Este un pointer spre un şir de caractere care defineşte calea spre fişierul care se deschide.

acces - Este o variabilă de tip întreg care poate avea una din valorile următoare:

- **O_RDONLY** - fişierul se deschide numai în citire (read-only);
- **O_WRONLY** - fişierul se deschide numai în scriere (write_only);
- **O_RDWR** - fişierul se deschide în citire/scriere;
- **O_APPEND** - fişierul se deschide pentru adăugarea de înregistrări la sfîrşitul lui;
- **O_BINARY** - fişierul se prelucrează binar;
- **O_TEXT** - fişierul este de tip text (se prelucrează pe caractere).

Aceste valori se pot combina cu ajutorul operatorului sau logic pe biţi ('|'). De exemplu:

O_RDWR|O_BINARY

înseamnă că fişierul este deschis în citire/scriere binară.

În mod implicit se consideră că un fişier este de tip text, deci O_TEXT se poate omite.

Utilizarea funcţiei *open* implică includerea fişierelor *io.h* şi *fcntl.h*:

```
#include <io.h>
```

şi

```
#include <fcntl.h>
```

Calea spre fişier trebuie să respecte convenţiile sistemului de operare MS-DOS. În cea mai simplă formă ea este un şir de caractere care defineşte numele fişierului, urmat de punct şi extensia fişierului. Aceasta presupune că fişierul respectiv se află în directorul curent.

În cazul în care fişierul nu este în directorul curent, numele este precedat de o construcţie de forma:

litera:*nume_1**nume_2*\...*nume_k*

unde:

litera - Defineşte discul (în general A, B pentru disc flexibil, C, D, ... pentru disc fix).

nume_i - Este nume de subdirector, i = 1,2,...,k.

Deoarece *calea* se include între ghilimele, caracterul '\' se dublează.

Deschiderea unui fişier nu reuşeşte dacă unul din parametri este eronat. În acest caz funcţia *open* returnează valoarea -1. Un caz frecvent de eroare este acela în care se încearcă deschiderea unui fişier inexistent.

Exemple:

1.

```
char nfis[] = "fis1.dat";
int df;
df = open(nfis,O_RDONLY);
```

Fişierul *fis1.dat* din directorul curent se deschide în citire. Funcţia returnează valoarea -1 dacă nu există un astfel de fişier în directorul curent.

2.

```
int d,
d = open("A:\\JOC\\BIO.C",O_RDWR);
```

Fişierul BIO.C din directorul JOC de pe discul A se deschide în citire/scriere. În acest caz şirul de caractere aflat pe locul primului

parametru al apelului funcţiei *open* se păstrează într-o zonă de memorie specială rezervată şirurilor de caractere. Adresa de început a acestei zone de memorie se atribuie primului parametru al funcţiei *open* în momentul apelului.

3.

```
int i;
i=open("c:\\borlandc\\include\\text\\text.h",O_APPEND);
```

Fişierul *text.h* se deschide pentru adăugare de înregistrări.

4.

```
int c;
c=open("fis.c",O_WRONLY);
```

În directorul curent trebuie să existe fişierul *fis.c*. Prin deschiderea lui în scriere, acesta se crează din nou, vechiul conţinut al fişierului pierzîndu-se.

Pentru a crea un fişier nou se utilizează funcţia *creat* în locul funcţiei *open*.

Aceasta are prototipul:

int creat(const char **calea*, **int** *mod*);

unde:

calea - Este un pointer spre un şir de caractere care defineşte calea spre fişierul care se deschide în creare.

mod - Este un întreg care poate fi definit folosind constantele simbolice de mai jos:

- **S_IREAD** - proprietarul poate citi fişierul;
- **S_IWRITE** - proprietarul poate scrie în fişier;
- **S_IEXE** - proprietarul poate executa programul conţinut în fişierul respectiv.

Aceşti indicatori pot fi combinaţi folosind caracterul '|'. De exemplu, pentru citire/scriere se va folosi:

```
S_IREAD|S_IWRITE
```

Funcţia *creat*, ca şi funcţia *open*, returnează descriptorul de fişier sau -1 în caz de eroare.

Utilizarea funcţiei *creat* implică includerile de fişiere:

```
#include <io.h>
```

şi

#include <stat.h>

În cazul în care funcţia *creat* se foloseşte pentru a deschide un fişier existent, atunci acesta se va şterge, urmînd ca în locul lui să se creeze fişierul nou.

Menţionăm că fişierele de intrare/ieşire standard se deschid automat la lansarea programului în execuţie. De aceea, aceste fişiere nu se deschid prin program de către programator.

Descriptorul de fişier reprezentat de intrarea standard are valoarea zero, iar cel reprezentat de ieşirea standard are valoarea unu.

Sistemul oferă utilizatorului încă o ieşire standard care, de obicei, este destinată pentru afişarea *erorilor*.

Fişierul corespunzător acestei ieşiri standard are descriptorul de fişier egal cu doi. Nici acest fişier nu se deschide prin program de către utilizator.

16.1.2. Citirea dintr-un fişier

Operaţia de citire a unei înregistrări dintr-un fişier deschis cu ajutorul funcţiei *open* se realizează folosind funcţia *read*. Aceasta returnează numărul de octeţi citiţi din fişier sau -1 la eroare.

Prototipul funcţiei este:

int read(int *df*, **void ****buf*, **unsigned** *lung*);

unde:

df - Este descriptorul de fişier returnat de funcţia *open* la deschiderea fişierul respectiv.

buf - Este pointerul spre zona de memorie în care se păstrează înregistrarea citită din fişier.

lung - Este lungimea în octeţi a înregistrării citite.

La fiecare apel al funcţiei *read* se citeşte înregistrarea curentă. Astfel, la primul apel se citeşte prima înregistrare din fişier, la al doilea apel a doua înregistrare şi aşa mai departe.

Ordinea înregistrărilor este cea definită la crearea fişierului şi eventual la adăugarea ulterioară de înregistrări noi.

La un apel al funcţiei *read* se citesc cel mult *lung* octeţi. La întîlnirea sfîrşitului de fişier nu se citeşte nimic şi de aceea funcţia *read* returnează zero în acest caz.

Dacă lung=1, atunci la apelul lui *read* se citeşte un singur octet din fişier. Acest lucru nu este avantajos mai ales atunci cînd se citesc înregistrări de pe disc. În acest caz o valoare utilizată frecvent pentru *lung* este 512.

Funcţia *read* poate fi folosită pentru a citi caractere de la intrarea standard. În acest caz descriptorul de fişier are valoarea zero la apelul funcţiei *read*.

Utilizarea funcţiei *read* implică includerea fişierului *io.h*

16.1.3. Scrierea într-un fişier

Pentru a scrie o înregistrare într-un fişier folosim funcţia *write*. Fişierul trebuie să fie în prealabil deschis cu ajutorul funcţiei *open* sau *creat*. Ea este asemănătoare cu funcţia *read* şi are acelaşi prototip. Diferenţa constă în aceea că realizează transferul datelor în sens invers, adică din zona de memorie spre care pointează cel de al doilea parametru al ei, în fişier.

Ea returnează numărul octeţilor scrişi în fişier, adică o valoare care coincide cu aceea a celui de al treilea parametru al ei. În cazul în care funcţia returnează o altă valoare decît valoarea celui de al treilea parametru din apel, înseamnă că scrierea a fost eronată.

Funcţia *write* poate fi folosită pentru a scrie la cele două ieşiri standard, folosind descriptori de fişier de valoare 1 sau 2. Utilizarea funcţiei *write* implică includerea fişierului *io.h*.

16.1.4. Poziţionarea într-un fişier

Operaţiile de citire/scriere se execută *secvenţial*. Aceasta înseamnă că la fiecare apel al funcţiei *read* sau *write* se citeşte înregistrarea curentă, respectiv se scrie înregistrarea în poziţia curentă de pe suportul fişierului.

Înregistrările se scriu una după alta pe suportul fişierului. La citire ele se citesc în aceeaşi ordine în care au fost scrise la crearea fişierului. Acest mod de scriere şi acces la înregistrările fişierului se numeşte *acces secvenţial*. Accesul secvenţial este util atunci cînd dorim să prelucrăm toate înregistrările fişierului, una după alta.

În practică apar însă situaţii în care noi dorim să scriem şi să citim înregistrări într-o ordine diferită de cea secvenţială, eventual chiar aleatoare. În acest caz se spune că *accesul* la fişier este *aleator*. Pentru a realiza un acces aleator este nevoie să ne putem poziţiona oriunde în fişierul respectiv. O astfel de poziţionare este posibilă pe suporturile de tip disc magnetic şi se realizează folosind funcţia *lseek*.

Ea are prototipul:

long lseek (int *df*, **long** *deplasament*, **int** *origine*);

unde:

df - Este descriptorul de fişier.

deplasament - Defineşte numărul de octeţi peste care se va deplasa capul de citire/scriere al discului.

origine - Are una din valorile:

- 0 - deplasamentul se consideră de la începutul fişierului;
- 1 - deplasamentul se consideră din poziţia curentă a capului de citire/scriere;
- 2 - deplasamentul se consideră de la sfîrşitul fişierului.

Prin apelul funcţiei *lseek* nu se realizează nici un fel de transfer de informaţie, ci numai poziţionarea în fişier. Operaţia următoare realizată prin apelul funcţiei *read* sau *write* se va realiza din această nouă poziţie a capului de citire/scriere.

Funcţia returnează poziţia capului de citire/scriere în număr de octeţi, faţă de începutul fişierului. În caz de eroare, se returnează -1L.

Utilizarea funcţiei *lseek* implică includerea fişierului *io.h*.

Apelul:

lseek(df,0l,2)

permite poziţionarea capului de citire/scriere la sfîrşitul fişierului definit de valoarea descriptorului de fişier *df*.

În mod analog, apelul:

lseek(df,0l,0)

permite poziţionarea capului de citire/scriere la începutul fişierului definit de *df*.

16.1.5. Închiderea unui fişier

După terminarea prelucrării unui fişier acesta trebuie închis. Acest lucru se realizează automat dacă programul se termină prin apelul funcţiei *exit*.

Programatorul poate închide un fişier folosind funcţia *close*. Se recomandă ca un fişier să se închidă de îndată ce s-a terminat prelucrarea lui. Aceasta din cauză că numărul fişierelor ce pot fi deschise simultan este

limitat. Această limită poate fi definită de utilizator. De obicei, ea are valoarea 20.

Menţionăm că fişierele corespunzătoare intrărilor şi ieşirilor standard nu se închid de către programator.

Funcţia *close* are prototipul:

int close (int *df*);

unde:

df - Este descriptorul fişierului care se închide.

La o închidere normală a fişierului, funcţia returnează valoarea zero. În caz de eroare se returnează -1.

Utilizarea funcţiei *close* implică includerea fişierului *io.h*.

Exerciţii:

16.1 Să se scrie un program care copiază intrarea standard la ieşirea standard folosind o zonă tampon de 70 de caractere.

PROGRAMUL BXVI1

```
#include <io.h>

#define LZT 70

main() /* copiaza intrarea standard la iesirea standard */
{
 char zt[LZT];
 int lung;

 while((lung=read(0,zt,LZT))>0)
     write(1,zt,lung);
}
```

16.2 Să se scrie un program care citeşte un şir de numere de la intrarea standard şi crează două fişiere *fis1.dat* şi *fis2.dat*, primul conţine numerele de ordin impar citite de la intrarea standard - primul, al treilea, al cincilea etc., iar al doilea conţine numerele de ordin par citite de la aceeaşi intrare - al doilea, al patrulea etc. Apoi se listează, la ieşirea standard, cele două fişiere în ordinea *fis1.dat*, *fis2.dat* cîte un număr pe un rînd în formatul:

număr de ordine număr

PROGRAMUL BXVI2

```
#include <stdio.h>
#include <stdlib.h>
#include <sys\stat.h>
#include <fcntl.h>
#include <io.h>
#include <conio.h>

main() /* - citeste un sir de numere si creaza fisierele fis1.dat cu nume-
              rele de ordin impar si fis2.dat cu numerele de ordin par;
            - apoi se listeaza fisierele. */
{
 union unr {
     float nr;
     char tnr[sizeof(float)];
 };
 union unr nrcit;
 int df1,df2;
 int i,j;

/* se deschid fisierele fis1.dat si fis2.dat in creare cu acces in
   citire/scriere */
 if((df1=creat("fis1.dat",S_IWRITE|S_IREAD))== -1){
     printf("nu se poate deschide fisierul\
           fis1.dat in creare\n");
     exit(1);
 }
  if((df2=creat("fis2.dat",S_IWRITE|S_IREAD))==
        -1){
     printf("nu se poate deschide fisierul\
           fis2.dat in creare\n");
     exit(1);
 }

/* se citeste sirul de numere si se pastreaza in cele doua fisiere */
 j=1;
 printf("tastati numerele separate prin caractere\
      albe\n");
  while ((scanf("%f",&nrcit.nr))==1){
     if(j&1){ /* j este impar */
         if((write(df1,nrcit.tnr,sizeof(float)))!=
             sizeof(float)){
            printf("eroare la crearea fisierului\
```

```
                fis1.dat\n");
            exit(1);
        }
    }
    else /* j este par */
        if((write (df2,nrcit.tnr,sizeof(float)))!=
             sizeof(float)){
            printf("eroare le crearea fisierului \
                  fis2.dat\n");
            exit(1);
        }
     j++;
}
```

/* s-a terminat citirea numerelor si scrierea lor in fisiere */

```
  if(close(df1)<0) {
    printf("eroare la inchiderea fisierului\
          fis1.dat\n");
    exit(1);
  }
  if(close(df2)<0) {
    printf("eroare la inchiderea fisierului\
          fis2.dat\n");
    exit(1);
  }
```

/* se redeschide fisierul fis1.dat si se listeaza la iesirea standard */

```
 if((df1=open("fis1.dat",O_RDONLY))== -1){
    printf("nu se poate deschide fisierul\
          fis1.dat in citire\n");
    exit(1);
 }
 j=1;
 while((i=read(df1,nrcit.tnr,sizeof(float)))>0){
    printf("%6d\t%g\n",j,nrcit.nr);
    if(j%46==0){
       printf("Actionati o tasta pentru a\
             continua\n");
       getch();
    }
    j += 2;
 }
 if(i<0){
```

```
        printf("eroare la citirea din fis1.dat\n");
        exit(1);
    }
    close(df1);
    printf("Actionati o tasta pentru a\
          continua \n\n\n");
    getch();

  /* se redeschide fisierul fis2.dat si se listeaza la iesirea standard */
    if((df2=open("fis2.dat",O_RDONLY))== -1){
        printf("nu se poate deschide fisierul\
              fis2.dat in citire\n");
        exit(1);
    }
    j=2;
    while((i=read(df2,nrcit.tnr,sizeof(float)))>0){
        printf("%6d\t%g\n",j,nrcit.nr);
        if(j%46==0){
            printf("Actionati o tasta pentru a\
                  continua\n");
            getch();
        }
        j += 2;
    }
    if(i<0){
        printf("eroare la citirea din fis2.dat\n");
        exit(1);
    }
    close(df2);
  }
```

16.3 Să se rescrie programul din exerciţiul precedent folosind un singur fişier *fis.dat*.

În acest caz se păstrează, în fişierul fis.dat, toate datele citite de la intrarea standard. Apoi fişierul *fis.dat* se parcurge de două ori, din două în două înregistrări, întîi începînd cu prima înregistrare, apoi începînd cu a doua.

PROGRAMUL BXVI3

```
#include <stdio.h>
#include <stdlib.h>
#include <sys\stat.h>
```

```
#include <fcntl.h>
#include <io.h>
#include <conio.h>

main()  /* - citeste un sir de numere si creaza fisierul fis.dat;
           - apoi listeaza fisierul fis.dat din doua in doua inregistrari;
           - intii se incepe cu prima inregistrare, apoi cu a doua. */
{
 union unr {
    float nr;
    char tnr[sizeof(float)];
 };
 union unr nrcit;
 int df;
 int i,j,k;

 /* se deschide fisierul fis.dat in creare cu acces in citire/scriere */
 if((df=creat("fis.dat",S_IWRITE|S_IREAD))== -1){
    printf("nu se poate deschide fisierul fis.dat\
          in creare\n");
    exit(1);
 }

/* se citeste sirul de numere si se pastreaza in fis.dat */
 printf("tastati numerele separate prin caractere\
       albe\n");
 while ((scanf("%f",&nrcit.nr))==1)
    if((write(df,nrcit.tnr,sizeof(float))) !=
        sizeof(float)){
       printf("eroare la crearea fisierului\
             fis.dat\n");
       exit(1);
    }

/* s-a terminat citirea numerelor si scrierea lor in fisier */
 if(close(df)<0) {
    printf("eroare la inchiderea fisierului\
           fis.dat\n");
    exit(1);
 }

/* se redeschide fisierul fis.dat si se listeaza la iesirea standard */
 if((df=open("fis.dat",O_RDONLY))== -1){
```

```
    printf("nu se poate deschide fisierul fis.dat\
          in citire\n");
    exit(1);
  }
  j=1;
  for(k=0;k<2;k++){
    while((i=read(df,nrcit.tnr,sizeof(float)))>0){
      printf("%6d\t%g\n",j,nrcit.nr);
      if(j%46==0){
        printf("Actionati o tasta pentru a\
              continua\n");
        getch();
      }
```

/* avans peste o inregistrare */

```
      if(lseek(df,(long)sizeof(float),1)== -1L)
        break;  /* s-a ajuns la sfirsitul fisierului fis.dat */
      j += 2;
    } /* sfirsit while */
    if(i<0){
      printf("eroare la citirea din fis.dat\n");
      exit(1);
    }
```

/* se trece la citirea a doua: se face pozitionare pe inregistrarea a doua a fisierului fis.dat */

```
    if(k<1){
      if(lseek(df,(long)sizeof(float),0)== -1L){
        printf("nu se poate face pozitionare pe\
              inregistrarea a doua\n");
        exit(1);
      }
      printf("Actionati o tasta pentru a\
            continua\n\n\n");
      getch();
      j=2;
    }
  } /* sfirsit for */
  close(df);
}
```

16.2. Nivelul superior de prelucrare a fişierelor

La acest nivel operaţiile de prelucrare a fişierelor se execută utilizîndu-se funcţii specializate de gestiune a fişierelor. De asemenea, fiecărui fişier i se ataşează o structură de tip FILE. Acest tip este definit în fişierul *stdio.h*. De asemenea, toate funcţiile din această clasă au prototipurile în fişierul *stdio.h*.

16.2.1. Deschiderea unui fişier

Pentru a deschide un fişier la acest nivel de prelucrare a fişierelor se utilizează funcţia *fopen*.

Ea returnează un pointer spre tipul FILE (tipul fişier) sau pointerul nul în caz de eroare.

Prototipul funcţiei este următorul:

FILE *fopen (const char *_calea_, const char *_mod_);

unde:

calea - Are aceeaşi semnificaţie ca şi în cazul funcţiei *open*;

mod - Este un pointer spre un şir de caractere care defineşte modul de prelucrare al fişierului după deschidere. Acest şir de caractere se defineşte astfel:

- "r" - deschidere în citire (read);
- "w" - deschidere în scriere (write);
- "a" - deschidere pentru adăugare (append);
- "r+" - deschidere pentru modificare (citire/scriere);
- "rb" - citire binară;
- "wb" - scriere binară;
- "r+b" - citire/scriere binară.

Cu ajutorul funcţiei *fopen* se poate deschide un fişier inexistent în modul *w* sau *a*. În acest caz, fişierul respectiv se consideră în creare.

Dacă se deschide un fişier existent în modul "w", atunci se va crea din nou fişierul respectiv şi vechiul conţinut al său se va pierde.

Deschiderea unui fişier în modul "a" permite adăugarea de înregistrări după ultima înregistrare existentă în fişier.

Cu ajutorul funcţiilor din această clasă se pot trata intrările şi ieşirile standard.

Fişierele corespunzătoare nu se deschid de către utilizator, deoarece ele sînt deschise automat la lansarea programului.

Pentru a utiliza aceste fişiere se vor folosi următorii pointeri spre tipul FILE:

stdin - Pentru a citi de la intrarea standard.

stdout - Pentru a afişa pe ecranul de la ieşirea standard.

stderr - Pentru afişarea erorilor la ieşirea standard.

stdprn - Ieşirea paralelă pe imprimantă.

stdaux - Comunicaţie serială.

16.2.2. Prelucrarea pe caractere a unui fişier

Fişierele pot fi scrise şi citite caracter cu caracter folosind două funcţii simple:

putc - Pentru scriere.

getc - Pentru citire.

Funcţia *putc* are prototipul:

int putc (int *c*, **FILE** ***pf**);

unde:

c - Este codul ASCII al caracterului care se scrie în fişier.

pf - Este un pointer spre tipul FILE a cărui valoare a fost returnată de funcţia *fopen* la deschiderea fişierului în care se face scrierea.
- În particular, *pf* poate fi unul din pointerii:
  - **stdout** (ieşire standard);
  - **stderr** (ieşire standard pentru eroare);
  - **stdprn** (ieşire paralelă la imprimantă);
  - **stdaux** (ieşire serială).

Funcţia *putc* returnează valoarea lui *c* sau -1 la eroare.

Macroul *putchar* se defineşte cu ajutorul funcţiei *putc* astfel:

#define putchar(*c***) putc(***c***,stdout)**

Această definiţie se află în fişierul *stdio.h*.

Funcţia *getc* are prototipul:

int getc (FILE *pf);

unde:

pf - Este pointerul spre tipul FILE.

Valoarea lui *pf* este definită la apelul funcţiei *fopen*.

În particular, *pf* poate fi unul din pointerii:

- **stdin** (intrare standard);
- **stdaux** (intrare serială).

Funcţia *getc* returnează codul ASCII al caracterului citit din fişier.

Macroul *getchar* se defineşte în fişierul *stdio.h* astfel:

#define getchar() getc(stdin)

Exerciţii:

16.4 Să se scrie un program care copiază intrarea standard la ieşirea standard folosind funcţiile *getc* şi *putc*.

PROGRAMUL BXVI4

```
#include <stdio.h>

main() /* copiaza intrarea standard la iesirea standard */
{
 int c;

 while((c=getc(stdin))!=EOF)
     putc(c,stdout);
}
```

Observaţie:

Programul poate fi scris şi cu ajutorul macrourilor *getchar* şi *putchar*. Nu avem decît să utilizăm:

getchar() în loc de *getc(stdin)*

şi

putchar(c) în loc de *putc(c,stdout)*

În urma expandării acestor macrouri se obţin chiar apelurile lui *getc* şi

putc din programul de mai sus.

16.2.3. Închiderea unui fişier

După terminarea prelucrării unui fişier, acesta urmează a fi închis. Închiderea unui fişier se realizează cu ajutorul funcţiei *fclose* de prototip:

int fclose (FILE *pf);

unde:

pf - Este pointerul spre tipul FILE. Valoarea lui a fost definită prin funcţia *fopen* la deschiderea fişierului.

Funcţia returnează valorile:

0 - La închidere normală.

-1 - La eroare.

Exerciţii:

16.5 Să se scrie un program care listează la imprimanta paralelă un fişier a cărui cale este argumentul din linia de comandă. În cazul în care în linia de comandă nu există un astfel de argument, se listează caracterele citite de la intrarea standard.

PROGRAMUL BXVI5

```
#include <stdio.h>
#include <stdlib.h>

main(int argc,char *argv[])
/*- listeaza la imprimanta paralela un fisier a carui cale este argumentul
    din linia de comanda;
  - daca argumentul lipseste, se listeaza caracterele de la intrarea
    standard. */
{
 FILE *pf;
 int c;

 if(argc ==1 )

/* nu exista argumente */
     pf=stdin;
 else
```

```
    if(argc==2){

/* se deschide fisierul a carui cale se afla in linia de comanda */
        if((pf=fopen(argv[1],"r"))==0){
            printf("nu se poate deschide fisierul\
                    %s\n",argv[1]);
            exit(1);
        }
    }
  else{
    printf("numar argumente eronat in linia\
            de comanda\n");
    exit(1);
  }
  while((c=getc(pf))!=EOF)
    putc(c,stdprn);
  fclose(pf);
}
```

16.2.4. Intrări/ieşiri de şiruri de caractere

Biblioteca standard a limbajului C conţine funcţiile *fgets* şi *fputs* care permit citirea, respectiv scrierea, înregistrărilor care sînt şiruri de caractere.

Funcţia *fgets* are prototipul:

char *fgets (char ****s*, int** *n*, **FILE ****pf***);**

unde:

s - Este pointerul spre zona în care se păstrează caracterele citite din fişier.

n - Este dimensiunea în octeţi a zonei în care se citesc caracterele din fişier.
- La un apel a lui *fgets* se citesc cel mult n-1 caractere.

pf - Este pointerul spre tipul FILE a cărui valoare s-a definit la deschiderea fişierului.

Citirea caracterelor se întrerupe la întîlnirea caracterului '\n' sau după citirea a cel mult n-1 caractere. În zona spre care pointează *s* se păstrează caracterul '\n' dacă acesta a fost citit din fişier, iar apoi se memorează caracterul nul ('\0').

În mod normal, funcţia returnează valoarea pointerului *s*. La întîlnirea sfîrşitului de fişier funcţia returnează valoarea zero.

Funcţia *fputs* scrie un şir de caractere (succesiune de caractere terminată cu '\0') într-un fişier.

Ea are prototipul:

int fputs (const char ***s*, FILE** ***pf*);

unde:

s - Este pointerul spre începutul zonei de memorie care conţine şirul de caractere care se scrie în fişier.

pf - Este pointerul spre tipul FILE a cărui valoare a fost definită la deschiderea fişierului prin apelul lui *fopen*.

Funcţia *fputs* returnează codul ASCII al ultimului caracter scris în fişier sau -1 la eroare.

Aceste două funcţii sînt similare cu funcţiile *gets* şi *puts* utilizate în multe din exerciţiile din această carte.

16.2.5. Intrări/ieşiri cu format

Biblioteca standard a limbajului C conţine funcţii care permit realizarea operaţiilor de intrare/ieşire cu format. În acest scop se pot utiliza funcţiile *fscanf* şi *fprintf*.

Acestea sînt similare cu funcţiile *sscanf*, respectiv *sprintf*. Diferenţa dintre ele constă în faptul că *fscanf* şi *fprintf* au ca prim parametru un pointer spre tipul FILE, iar *sscanf* şi *sprintf* au ca prim parametru un pointer spre o zonă în care se păstrează caractere. De aici rezultă şi utilizările celor două funcţii. Astfel, funcţia *fscanf* citeşte date dintr-un fişier şi le converteşte păstrînd rezultatele acestor conversii în conformitate cu parametri existenţi la apelul funcţiei, în timp ce *sscanf* realizează acelaşi lucru dar utilizînd date din memorie. În mod analog, funcţia *fprintf*, converteşte date din format intern în format extern şi apoi le scrie într-un fişier, spre deosebire de funcţia *sprintf* care realizează aceleaşi conversii, dar rezultatele se păstrează în memorie.

Prototipul funcţiei *fscanf* este:

int fscanf (FILE ***pf*, **const char** ***format*,...);

unde:

pf - Este un pointer spre tipul FILE a cărui valoare a fost definită prin apelul funcţiei *fopen*.
- Acesta defineşte fişierul din care se face citirea.

Ceilalţi parametri sînt identici cu cei utilizaţi în funcţia *scanf*.

Funcţia *fscanf*, ca şi funcţiile *scanf* şi *sscanf*, returnează numărul cîmpurilor citite corect din fişier. La întîlnirea sfîrşitului de fişier funcţia returnează valoarea EOF, definită în fişierul *stdio.h*.

Funcţia *fprintf* are prototipul:

int fprintf (**FILE** **pf*, **const char** **format*,...);

unde:

pf - Este un pointer spre tipul FILE a cărui valoare a fost definită prin apelul funcţiei *fopen*.
- Acesta defineşte fişierul în care se face scrierea.

Ceilalţi parametri sînt identici cu cei utilizaţi în funcţia *printf*.

Funcţia *fprintf*, ca şi funcţiile *printf* şi *sprintf*, returnează numărul caracterelor scrise în fişier sau -1 în caz de eroare.

16.2.6. Vidarea zonei tampon a unui fişier

Există situaţii în care programatorul doreşte să videze zona tampon a unui fişier. În acest scop se poate utiliza funcţia *fflush*. Ea are prototipul:

int fflush (**FILE** **pf*);

unde:

pf - Este pointerul spre tipul FILE care defineşte fişierul pentru care se videază zona tampon.

Dacă fişierul este deschis în scriere, atunci conţinutul zonei tampon se scrie în fişierul respectiv.

Dacă fişierul este deschis în citire, caracterele necitite din zona tampon se pierd, deoarece după apelul funcţiei zona tampon devine goală.

Un exemplu de utilizare a acestei funcţii este acela al recuperării după o eroare în citire.

Exemplu:

```
int i,n;
...
do {
 printf("tastati valoare lui n=");
 if((i=scanf("%d",&n))==1)
```

```
        break;
    printf("nu s-a tastat un intreg\n");
    if(i==EOF) /* s-a tastat sfirsitul de fisier */
        exit(1);
    fflush(stdin); /* se elimina caracterele necitite din zona
                      tampon atasata intrarii standard */
}while(1);
```

Funcţia returnează valoarea -1 în caz de eroare şi zero dacă vidarea se realizează fără erori.

16.2.7. Poziţionarea într-un fişier

Biblioteca standard a limbajului C conţine funcţia *fseek* care permite deplasarea capului de citire/scriere al discului în vederea prelucrării înregistrărilor fişierului într-o ordine diferită de cea secvenţială.

Ea are prototipul:

int fseek (**FILE** **pf*, **long** *deplasament*, **int** *origine*);

unde:

pf - Este pointerul spre tipul FILE care defineşte fişierul în care se face poziţionarea capului de citire/scriere.

Valoarea lui se defineşte la deschiderea fişierului prin apelul funcţiei *fopen*.

Ceilalţi parametri se definesc la fel ca în cazul funcţiei *lseek*.

Funcţia *fseek* returnează valoarea zero la o poziţionare corectă şi o valoare diferită de zero în caz de eroare.

O altă funcţie utilă în cazul accesului aleator la fişier, este funcţia *ftell*. Această funcţie permite să se cunoască poziţia curentă a capului de citire/scriere. Ea are prototipul:

long ftell (**FILE** **pf*);

unde:

pf - Este pointerul spre tipul FILE care defineşte fişierul în cauză.

Funcţia returnează valoarea care defineşte poziţia curentă a capului de citire/scriere.

Valoarea returnată de *ftell* este deplasamentul în octeţi a poziţiei capului de citire/scriere faţă de începutul fişierului.

16.2.8. Prelucrarea fişierelor binare

Fişierele organizate ca date binare (octeţii nu sînt consideraţi ca fiind coduri de caractere) pot fi prelucrate folosind funcţiile *fread* şi *fwrite*.

În acest caz se consideră că înregistrarea este o colecţie de *articole*. Articolul este o dată de un tip oarecare (predefinit sau definit de utilizator). La o citire, se transferă într-o zonă specială, numită *zonă tampon* (buffer), un număr de articole care se presupune că au o lungime fixă. În mod analog, la scriere se transferă din zona tampon un număr de articole de lungime fixă.

Prototipul funcţiei *fread* este următorul:

unsigned fread (void ***ptr*****, unsigned** *dim***, unsigned** *nrart***, FILE** ***pf*****);**

unde:

ptr	- Este pointerul spre zona tampon care conţine articolele citite (înregistrarea citită).
dim	- Defineşte dimensiunea unui articol în octeţi.
nrart	- Defineşte numărul de articole din compunerea înregistrării citite.
pf	- Este pointerul spre tipul FILE care defineşte fişierul din care se face citirea.

Funcţia returnează numărul de articole citite sau -1 în caz de eroare.

Funcţia *fwrite* are prototipul similar cu cel al funcţiei *fread*:

unsigned fwrite (void ***ptr*****, unsigned** *dim***, unsigned** *nrart***, FILE** ***pf*****);**

unde:

ptr	- Este pointerul spre zona tampon care conţine articolele care se scriu în fişier.
dim	- Defineşte lungimea unui articol.
nrart	- Defineşte numărul de articole din compunerea înregistrării care se scrie în fişier.
pf	- Este pointerul spre tipul FILE care defineşte fişierul în care se scrie înregistrarea.

Funcţia returnează numărul articolelor scrise în fişier sau -1 în caz de eroare.

Menţionăm că pentru a prelucra fişiere binare, la deschiderea lor se foloseşte modul *b*:

"wb"	- Pentru creare.

"rb" - Pentru citire.
"r+b" - Pentru citire/scriere.
"w+b" - Idem.
"ab" - Pentru adăugare de înregistrări.

16.3. Ştergerea unui fişier

Un fişier poate fi şters apelînd funcţia *unlink*. Aceasta are prototipul:

int unlink (const char ****calea*);**

unde:

calea - Este un pointer spre un şir de caractere identic cu cel utilizat la crearea fişierului în funcţia *creat* sau *fopen*.

Funcţia *unlink* returnează valoarea zero la o ştergere reuşită şi -1 în caz de eroare.

16.4. Redirectarea fişierelor de intrare/ieşire standard

Am văzut că funcţiile: *scanf*, *printf*, *gets* şi *puts*, precum şi macrourile *getchar* şi *putchar* citesc şi scriu la terminalul standard. Ele pot fi utilizate şi cu alte periferice dacă, în prealabil, se face o redirectare a fişierelor standard de intrare/ieşire. Acest lucru se realizează folosind, în linia de comandă a apelului execuţiei programului, caracterele '<'şi'>'.

Caracterul '<' se foloseşte pentru redirectarea fişierului de intrare (*stdin*), iar caracterul '>' pentru redirectarea fişierului de ieşire (*stdout*).

Aceste caractere se folosesc în felul următor:

< *specificator_de_fisier_de_intrare*

şi

> *specificator_de_fisier_de_iesire*

Specificatorul de fişier depinde de sistemul de operare utilizat.

În cazul sistemului de operare MS-DOS specificatorul de fişier poate fi:

prn - Pentru ieşirea pe imprimanta paralelă.
com1 - Pentru transferul de date serial.
calea - Spre un fişier de pe disc.

NUL - Perifericul nul.

Menţionăm că fişierul de ieşire standard pentru erori (*sderr*) nu poate fi redirectat.

Exemple:

1. Programul BXVI4 copiază intrarea standard la ieşirea standard. Fie BXVI4.EXE imaginea executabilă a programului respectiv.

   Un apel de forma:

   >BXVI4.EXE >prn

   afişează la imprimanta paralelă caracterele citite de la intrarea standard.

2. Apelul:

   >BXVI4.EXE <fis1.dat >fis2.dat

   crează fişierul *fis2.dat* prin copierea fişierului *fis1.dat*. Ambele fişiere sînt în directorul curent (în care se află şi fişierul BXVI4.EXE).

3. Apelul:

   >BXVI4.EXE <fis1.dat

   afişează la ieşirea standard conţinutul fişierului *fis1.dat*.

Menţionăm că pentru a opri afişarea în vederea analizării datelor afişate, se acţionează tasta *Break*. Pentru a continua afişarea se va acţiona o tastă oarecare, de exemplu bara pentru spaţiu.

Exerciţii:

16.6 Să se scrie un program care citeşte de la intrarea standard datele a căror format sînt definite mai jos şi le scrie în fişierul misc.dat din directorul curent.

Fişierul este organizat binar, fiecare articol conţinînd următoarele date:

tip	denumire	um	cod	preţ	cantitate
I	REZISTENTA	010KO	123456789	10	10000

PROGRAMUL BXVI6

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 50
```

```
typedef struct {
 char tip[2];
 char den[MAX+1];
 int val;
 char unit[3];
 long cod;
 float pret;
 float cant;
} ARTICOL;
/* 6 articole in zona tampon */

union {
 ARTICOL a[6];
 char zt[6*sizeof(ARTICOL)];
}buf;

int citart(ARTICOL *str)
/* - citeste datele de la intrarea standard si le pastreaza in zona spre care
     pointeaza str;
   - returneaza EOF la intilnirea sfirsitului de fisier si 1 in caz contrar. */
{
 int c,nr;
 float x,y;

 for(;;)
     if((nr=scanf("%1s %50s %3d %2s %ld %f %f",
           str->tip,str->den,&str->val,str->unit,
            &str->cod,&x,&y))!=7){
          if(nr==EOF)
              return EOF;
          printf("date eronate;se reia citirea\n");
          fflush(stdin);
     }
     else
          break;
 str->pret=x;
 str->cant=y;
 return 1;
} /* sfirsit citart */

main() /* crearea fisierului misc.dat cu datele citite de la intrarea
          standard */
```

```
{
 FILE *pf;
 int i,n;

/* se deschide fisierul in scriere binara */
 if((pf=fopen("misc.dat","wb"))==0){
     printf("nu se poate deschide in creare\
           misc.dat\n");
     exit(1);
 }
 for(;;){

/* se umple zona tampon a fisierului */
     for(i=0;i<6;i++)
         if((n=citart(&buf.a[i]))==EOF)
             break; /* s-a intilnit EOF */

/* se scrie zona tampon daca nu este vida */
     if(i) /* zona tampon nu este vida */
         if(i!=fwrite(buf.zt,sizeof(ARTICOL),i,pf)){

/* eroare la scrierea in fisier */
             printf("eroare la scrierea in\
                    fisier\n");
             exit(1);
         }
     if(n==EOF)
         break;
 } /* sfirsit for */
 fclose(pf);
} /* sfirsit main */
```

Observaţie:

În datele de intrare, *tip* este o literă:

I - Intrări.

E - Ieşiri.

T - Transfer etc.

Tipul se citeşte folosind specificatorul %1s. În felul acesta se omit eventualele caractere albe care preced litera ce defineşte tipul.

16.7 Să se scrie un program care afişează articolele păstrate în fişierul misc.dat creat în exerciţiul precedent, pentru tip=I.

PROGRAMUL BXVI7

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 50

typedef struct {
 char tip[2];
 char den[MAX+1];
 int val;
 char unit[3];
 long cod;
 float pret;
 float cant;
}ARTICOL;

union {
 ARTICOL a[6];
 char zt[6*sizeof(ARTICOL)];
}buf;

main() /* citeste si afiseaza articolele din fisierul misc.dat
          pentru tip=I */
{
 FILE *pf;
 int i,n;

 if((pf=fopen("misc.dat","rb"))==0){
     printf("nu se poate deschide fisierul\
           misc.dat\n");
     exit(1);
 }
 printf("%30s\n\n\n","INTRARI");
 while((n=fread(buf.zt,sizeof(ARTICOL),6,pf))>0)

/* listeaza articolele dintr-o inregistrare cu tipul=I */
     for(i=0;i<n;i++)
         if(buf.a[i].tip[0]=='I')
            printf("%-50s %03d%s %091d %.2f %.2f\n",
                  buf.a[i].den,buf.a[i].val,
```

```
                        buf.a[i].unit,buf.a[i].cod,
                        buf.a[i].pret,buf.a[i].cant);
 fclose(pf);
}
```

17. FUNCŢII STANDARD

Mediul de programare TURBO C pune la dispoziţia utilizatorului un set de funcţii standard care pot fi utilizate cu succes într-o serie de aplicaţii din diferite domenii. Aceste funcţii pot fi apelate atît din programe scrise în C cît şi din cele scrise în C++. Ele pot fi grupate în mai multe clase, ţinînd seama de utilizările lor.

Apelurile funcţiilor standard implică includeri de fişiere de tip *h* care conţin prototipurile lor. Ca exemple de astfel de fişiere putem aminti: *stdio.h*, *stdlib.h*, *conio.h*, *io.h*, *fcntl.h*, *alloc.h* etc.

De obicei, fişierele de tip *h* relative la funcţiile standard se află în subdirectorul INCLUDE al directorului TC pentru TURBO C sau BORLANDC pentru TURBO C++.

Listarea subdirectorului INCLUDE printr-o comandă de forma:

dir \tc\include

permite afişarea numelor fişierelor de tip *h*.

În continuare se poate lista conţinutul fişierului *nume.h* folosind o comandă de forma:

type \tc\include\nume.h

În cazul în care dorim să afişăm la imprimanta paralelă conţinutul fişierului *nume.h*, putem utiliza comanda de mai jos:

type \tc\include\nume.h >prn

Printr-o astfel de listare se pot pune în evidenţă prototipurile funcţiilor standard aflate în fişierul respectiv.

Pentru a obţine informaţii suplimentare despre o funcţie standard se poate proceda ca mai jos.

1. Setare pe directorul *tc* sau *borlandc*:

   >cd \tc sau cd \borlandc

2. Se activează mediul de dezvoltare integrat TURBO C sau C++ tastînd:

   tc sau bc

3. Se activează meniul de editare:

   <Alt> - E

4. Se tastează, în fereastra de editare, numele funcţiei standard despre

care se doresc informaţii.

5. Acţionînd tasta cu săgeată spre stînga, se vine cu cursorul pe una din literele din compunerea numelui funcţiei respective.

6. Se tastează

   <Ctrl> - F1

Se obţine o fereastră de *Help* cu explicaţii despre funcţia respectivă. Aceste explicaţii conţin:

- prototipul funcţiei;
- fişierul sau fişierele de tip *h* care conţin prototipul funcţiei respective;
- descrierea pe scurt a efectului apelului funcţiei respective;
- nume de parametri şi eventualele funcţii înrudite pentru care se pot consulta informaţii de *help* care să vină în completarea celor de faţă;
- exemple de utilizare a funcţiei respective.

Cu ajutorul tastelor cu săgeţi şi a tastei TAB se pot selecta nume de parametri şi de funcţii înrudite în vederea obţinerii de informaţii de *help* suplimentare. În acest scop, după selectarea unui astfel de nume se acţionează tasta *Enter*. Se deschide o nouă fereastră de *help*.

În cazul în care există mai multe ferestre de *Help*, se poate trece de la o fereastră la alta folosind tasta *PgDn*.

Pentru a reveni la o fereastră anterioară vom acţiona tasta *PgUp*.

Pentru a reveni dintr-o fereastră de *Help* în fereastra de editare se acţionează tasta *ESC*.

Amintim că pentru a reveni în sistemul de operare se acţionează:

<Alt> - X

În cele ce urmează vom trece în revistă clasele de funcţii standard utilizate mai frecvent.

O primă clasă de funcţii sînt cele destinate prelucrării fişierelor. Funcţiile din această clasă care se utilizează mai frecvent au fost descrise în capitolul precedent. De asemenea, funcţiile utilizate la operaţiile de intrare/ieşire cu terminalul standard au fost folosite în toate capitolele cărţii de faţă. De aceea, în acest capitol nu ne vom mai referi la funcţiile din această clasă.

O altă clasă de funcţii standard se referă la alocarea dinamică a memoriei. Acestea au prototipul în fişierul *alloc.h*.

Funcţiile mai importante din această clasă au fost descrise şi utilizate deja în mai multe capitole din carte. În acest capitol ne vom referi la încă o funcţie din această clasă care uneori are o utilizare importantă (funcţia *coreleft*).

Într-un capitol precedent au fost prezentate funcţiile mai importante utilizate la prelucrarea şirurilor: *strlen*, *strcpy*, *strcat* şi *strcmp* împreună cu unele versiuni ale lor.

Ele au prototipul în fişierul *string.h*.

Aceste funcţii au fost utilizate în diferite exerciţii din capitolele precedente şi de aceea nu ne vom mai referi la ele în capitolul de faţă.

O clasă importantă de funcţii o constituie funcţiile de gestiune a ecranului. Aceste funcţii au prototipul în fişierele *conio.h* şi *graphics.h*. Ele prezintă o importanţă mare în diferite aplicaţii şi de aceea funcţiile respective vor fi descrise şi aplicate în capitolele următoare.

În capitolul de faţă ne vom opri asupra următoarelor clase:

- funcţii pentru realizarea de conversii;
- funcţii de calcul;
- funcţii pentru gestiunea memoriei;
- funcţii de control ale proceselor;
- funcţii de gestiune a datei şi orei.

De asemenea, ne vom opri şi asupra unor macrouri definite în fişierul *ctype.h* şi anume:

macrouri de clasificare

şi

macrouri de transformare a simbolurilor.

17.1. Macrouri de clasificare

În această clasă distingem un număr de macrouri care au aplicaţii în prelucrarea caracterelor. Aceste macrouri sînt definite în fişierul *ctype.h*.

Un prim macro din această clasă este macroul *isascii* de prototip:

int isascii(int *c*);

El returnează o valoare diferită de zero dacă valoarea lui *c* aparţine intervalului [0,127] şi zero în caz contrar.

Celelalte macrouri sînt de forma:

int is ... (int *c*);

Valoarea lui *c* trebuie să aparţină intervalului [0,127]. Cele trei puncte reprezintă o succesiune de litere ca mai jos.

Aceste macrouri reprezintă o valoare diferită de zero dacă *c* este:

pentru	**isalpha**	o literă
	isalnum	o literă sau o cifră
	isdigit	o cifră
	isgraph	un caracter grafic
	islower	o literă mică
	isprint	un caracter imprimabil
	isspace	un caracter spaţiu, tabulator, retur de car, interline, tabulator vertical sau salt de pagină
	isupper	o literă mare
	isxdigit	o cifră hexazecimală

17.2. Macrouri de transformare a caracterelor

În această clasă distingem macrouri definite tot în fişierul *ctype.h*.

Prototipurile lor sînt:

int toascii(int *c*); - returnează ultimii 7 biţi ai lui *c*.
int tolower (int *c*); - transformă pe *c* din literă mare în literă mică.
int toupper(int *c*); - transformă pe *c* din literă mică în literă mare.

Exemplu:

Modificăm programul din exerciţiul 16.4 aşa încît literele mari de la intrarea standard să fie afişate ca litere mici:

```
...
while((c = getchar()) != EOF)
 if(isupper(c))
    putchar(tolower(c));
 else
    putchar(c);
...
```

17.3. Funcţii de conversie

Biblioteca standard a limbajului C conţine mai multe funcţii de conversie a datelor din format extern în cel intern şi invers.

Pînă în prezent am utilizat două astfel de funcţii:

sscanf

şi

sprintf

Aceste funcţii au prototipurile în fişierul *stdio.h*. Ele permit realizarea de conversii sub controlul formatelor.

În afară de aceste două funcţii, biblioteca standard mai conţine şi alte funcţii de conversie dintre care amintim pe cele utilizate mai frecvent.

Funcţia *atoi* are prototipul:

int atoi(const char ***p);**

unde:

p	- Pointează spre un şir de caractere în compunerea căruia intră cifre zecimale şi care eventual sînt precedate de un semn.

Funcţia converteşte întregul zecimal definit de şirul spre care pointează *p* în binar de tip *int* şi returnează rezultatul acestei conversii.

Funcţia *itoa* realizează conversia inversă.

Ea are prototipul:

char *itoa(int *val*, **char ***sir*, **int** *baza*);

unde:

val	- Este valoarea binară de tip *int* care se converteşte ca un întreg în baza de numeraţie definită de *baza*.
sir	- Este pointerul spre zona în care se păstrează rezultatul conversiei sub formă de şir de caractere.
baza	- Este baza rezultatului.

Funcţia returnează valoarea pointerului *sir*.

Biblioteca conţine funcţii similare pentru tipul *long*. Astfel, funcţia *atol* are prototipul:

long atol(const char ***p);**

şi realizează conversia din zecimal a şirului spre care pointează *p*, în binar de tip *long*.

Conversia inversă se realizează cu ajutorul funcţiei *ltoa*:

char *ltoa(long *val*,**char** *****sir*,**int** *baza*);

Parametrii *val,sir* şi *baza* au aceleaşi utilizări ca şi în cazul funcţiei *itoa*. Funcţia returnează valoarea pointerului *sir*.

Toate aceste funcţii au prototipul în *stdlib.h*.

Tot din această clasă face parte şi funcţia *atof* de prototip:

double atof(const char *****p*);

Această funcţie converteşte un număr aflat în zona spre care pointează *p*, în format flotant dublă precizie şi returnează rezultatul acestei conversii. Funcţia *atof* realizează aceeaşi conversie ca şi cea realizată cu ajutorul funcţiei *scanf* relativ la specificatorul de format %lf. Aceasta înseamnă că în compunerea numărului spre care pointează *p* se poate afla caracterul punct, precum şi un exponent.

Funcţia *atof* are prototipul în fişierele *math.h* şi *stdlib.h*.

17.4. Funcţii de calcul

Majoritatea funcţiilor standard utilizate la calcule numerice au prototipul în fişierul *math.h*. Multe dintre acestea au prototipul:

double *nume* (**double** *x*);

unde:

nume - Are semnificaţia de mai jos:

nume	**echivalentul în limbajul matematic**
acos	arccos
asin	arcsin
atan	arctg
cos	cos
sin	sin
exp	e la puterea x
log	ln
log10	log în baza 10
sqrt	rădăcina pătrată
pow10	10 la puterea x
ceil	rotunjire prin exces
floor	rotunjire prin lipsă
fabs	valoare absolută

sinh	sh
cosh	ch
tanh	th

Alte funcţii de calcul cu prototipurile în *math.h* şi care se utilizează frecvent sînt:

double atan2(double *y*,**double** *x*);

returnează arctg(*y/x*);

double pow(double *x*,**double** *y*);

returnează *x* la puterea *y*;

double cabs(struct complex *z*);

returnează modulul numărului complex *z*;

double poly(double *x*, **int** *n*, **double** *c[]*);

returnează valoarea polinomului în *x* de grad *n*, coeficienţii fiind păstraţi în tabloul *c* astfel:

c[0] - Termen liber.

c[1] - Coeficientul lui *x*.

...

c[i] - Coeficientul lui *x* la puterea *i*.

...

c[n] - Coeficientul lui *x* la puterea *n*.

Următoarele funcţii de calculul au prototipurile în fişierul *stdlib.h*:

int abs(int *x*);

returnează valoarea absolută a lui *x* de tip *int*;

long labs(long *x*);

returnează valoarea absolută a lui *x* de tip *long*;

int rand(void);

returnează un număr natural pseudo-aleator mai mic decît 32768;

int random(int *nr*);

returnează un număr natural pseudo-aleator mai mic decît *nr*;

void srand(int *n*);

setează sămînţa numerelor pseudo-aleatoare la valoare lui *n*.

O parte din funcţiile de calcul amintite mai sus au fost utilizate în diferite exerciţii din capitolele acestei cărţi.

17.5. Funcţii pentru gestiunea memoriei

Funcţiile din această clasă au prototipurile în fişierul *alloc.h*.

În paragraful 8.8 au fost amintite funcţiile *malloc, calloc, free,* şi versiunile acestora pentru cazul în care pointerii sînt de tip *far* : *farmalloc, farcalloc* şi *farfree*.

În acest paragraf amintim funcţia *coreleft* care are prototipul:

unsigned coreleft (void);

Această funcţie se utilizează pentru modelele de memorie: *tiny, small* şi *medium* (pentru aceste modele pointerii sînt de tip *near*). Ea returnează dimensiunea memoriei libere în momentul apelului ei.

Pentru modelele de memorie *compact, large* şi *huge* (pointerii sînt de tip *far*) funcţia *coreleft* are prototipul:

unsigned long coreleft(void);

Menţionăm că în cazul mediilor integrate de dezvoltare Turbo C şi C++, modelul se defineşte cu ajutorul submeniului *Compile* al meniului *Options*.

17.6. Funcţii de control ale proceselor

Indicăm mai jos prototipurile unor funcţii mai importante din această clasă.

Funcţia *exit* are prototipul:

void exit(int *stare***)**;

Ea termină execuţia programului.

Parametrul *stare* are valoarea zero la o terminare normală a execuţiei programului şi diferită de zero pentru o terminare anormală. Funcţia *exit* videază zonele tampon ale fişierelor deschise în scriere, în momentul apelului ei. Apoi se închid toate fişierele deschise.

Funcţia *system* are prototipul:

int system(const char **comanda***)**;

unde:

comanda - Este un pointer spre un şir de caractere care reprezintă o comandă MS-DOS. Ea are ca efect execuţia comenzii definite de şirul de caractere spre care pointează parametrul *comanda*.

Funcţia returnează codul de stare al execuţiei comenzii:

0 - La terminare normală.

-1 - La eroare.

Funcţia *spawnl* este asemănătoare cu funcţia *system*. Ea permite lansarea în execuţie a imaginii executabile a unui program păstrat într-un fişier.

Prototipul funcţiei este:

int spawnl(int *mod*,**char** ***cale**,**char** **arg0*, **char** **arg1*,...,**NULL);**

unde:

mod - Are ca valoare constanta simbolică P_WAIT.

- Această valoare înseamnă că programul din care se apelează funcţia *spawnl* îşi întrerupe execuţia pînă la terminarea execuţiei programului lansat prin apelul funcţiei *spawnl*.

cale - Este pointerul spre şirul de caractere care defineşte fişierul cu imaginea executabilă a programului care se lansează.

arg0,arg1,... - Sînt pointeri spre şiruri de caractere care reprezintă argumentele programului care se lansează.

După o execuţie cu succes a programului apelat, funcţia *spawnl* returnează valoarea codului de stare de la apelul funcţiei *exit* prin care s-a terminat execuţia programului respectiv.

În caz de eroare (nu se reuşeşte lansarea în execuţie a unui program), funcţia *spawnl* returnează valoarea -1.

Aceste funcţii au prototipul în fişierele *stdlib.h* şi *process.h*.

17.7. Funcţii pentru gestiunea datei şi a orei

Indicăm mai jos prototipurile a patru funcţii pentru citirea/setarea datei şi a orei. Ele implică includerea fişierului *dos.h*.

Funcţiile pentru citirea datei şi orei au prototipurile:

void getdate(struct date **d*);

şi

void gettime(struct time ***t*****);**

Funcţiile pentru setarea datei şi orei au prototipurile:

void setdate(struct date ***d*****);**

void settime(struct time ***t*****);**

Structurile *date* şi *time* sînt definite în *dos.h* astfel:

```
struct date {
        int da_year;
        int da_day;
        int da_mon;
};
struct time {
        unsigned char ti_min;
        unsigned char ti_hour;
        unsigned char ti_hund;
        unsigned char ti_sec;
};
```

17.8. Diferite funcţii de uz general

Funcţia *clrscr* de prototip:

void clrscr(void);

şterge fereastra activă sau tot ecranul dacă n-a fost activat în prealabil o fereastră (definirea şi activarea unei ferestre se va descrie într-un capitol următor; în mod implicit *clrscr* şterge tot ecranul).

Prototipul funcţiei se află în fişierul *conio.h*.

Funcţia *kbhit* de prototip:

int kbhit(void);

returnează o valoare diferită de zero dacă există un caracter disponibil de la tastatură; în caz contrar ea returnează valoarea zero.

Funcţia *kbhit* se poate apela ciclic pentru a aştepta tastarea unui caracter la terminalul standard. Astfel, ciclul:

```
while(!kbhit())
;
```

se execută pînă în momentul în care se acţionează o tastă la terminalul

standard.

Pe perioada execuţiei ciclului se afişează ecranul utilizator şi de aceea acest ciclu se poate utiliza la fel ca şi apelul funcţiei *getch* pentru a afişa ecranul respectiv.

Funcţiile care urmează şi încep cu prefixul *str* implică includerea fişierului *string.h*.

Funcţia *strlwr* are prototipul:

char *strlwr(char *_sir_);

Ea converteşte literele mari din zona spre care pointează *sir* în litere mici.

Funcţia returnează valoarea *sir*.

Funcţia *strupr* de prototip:

char *strupr(char *_sir_);

este inversa funcţiei *strlwr*.

Funcţia *strset* de prototip:

char *strset(char *_sir_,int _c_);

iniţializează zona spre care pointează *sir* cu valoarea lui *c* (ultimii 8 biţi, cei mai puţini semnificativi). Substituirea valorii lui *c* se realizează pînă la întîlnirea caracterului NUL, care rămîne nemodificat. Funcţia returnează valoarea lui *sir*.

O variantă a acestei funcţii este funcţia *strnset*. Ea are prototipul:

char *strnset(char *_sir_,int _c_,unsigned _n_);

Funcţia are aceeaşi acţiune ca şi funcţia *strset*, cu deosebirea că se iniţializează cu valoare lui *c* numai cel mult primele *n* caractere ale zonei spre care pointează *sir*.

Funcţia *strchr* are prototipul:

char *strchr(const char *_sir_,char _c_);

Se caută prima apariţie a caracterului *c* în zona spre care pointează *sir*.

Funcţia returnează pointerul spre poziţia determinată sau zero în cazul în care nu există o intrare a caracterului *c* în zona respectivă.

Funcţia *strrchr* are prototipul:

char *strrchr(const char *_sir_,char _c_);

Ea are aceeaşi acţiune ca şi funcţia *strchr*, cu deosebirea că determină nu prima apariţie a caracterului *c*, ci ultima.

Funcţiile care urmează au prototipul în fişierul *dos.h*.

Funcţia *delay* are prototipul:

void delay(unsigned *i*);

Ea suspendă execuţia programului pentru o perioadă de *i* milisecunde.

Funcţia *sleep* are prototipul:

void sleep(unsigned *i*);

Ea suspendă execuţia programului pentru o perioadă de *i* secunde.

Funcţia *sound* are prototipul:

void sound(unsigned *h*);

Ea porneşte difuzorul calculatorului cu un ton egal cu *h* Hz.

Funcţia *nosound* are prototipul:

void nosound(*void*);

Ea opreşte sunetul de la difuzorul calculatorului.

17.9. Tratarea erorilor

Distingem două mari categorii de erori:

erori care apar la compilare;

şi

erori care apar la execuţie.

Erorile de compilare sînt de trei nivele:

erori fatale
- Semnifică o eroare în compilator.
- Dacă aceste erori nu provin de la apelul incorect al unei definiţii de macro, atunci ele vor fi semnalate firmei Borland.

erori
- Compilatorul afişează, în fereastra de mesaje, pentru fiecare eroare, un text explicit al acesteia.
- Sînt prevăzute aproximativ 140 de cazuri de eroare.

avertismente
- Sînt mesaje care indică:
  - Situaţii care în anumite cazuri pot constitui erori.

- Informaţii cu privire la portabilitatea şi calitatea programului. Sînt prevăzute circa 30 de astfel de cazuri.

Erorile care apar la execuţie pot fi tratate prin:

- Valorile returnate de DOS prin intermediul variabilelor globale *errno, doserrno, sys_errlist, sys_nerr*. Aceste variabile sînt declarate în fişierele <errno.h> şi <dos.h>. Funcţia *perror* afişează mesajul de eroare corespunzător.
- Gestiunea erorilor prin funcţia *harderr* din <dos.h>.
- Detecţia erorilor prin funcţia *ferror* în prelucrarea fişierelor.
- Gestiunea erorilor de calcul prin apelul funcţiei *matherr*.

Exerciţii:

17.1 Programul de mai jos afişează data şi ora furnizate de sistemul de operare. Formatul de afişare este:

zz/ll/aaaa ora hh:mm:ss

PROGRAMUL BXVII1

```
#include <stdio.h>
#include <dos.h>

main() /* afiseaza data si ora */
{
 struct date d;
 struct time t;

 getdate(&d);
 gettime(&t);
 printf("\n\t%02d/%02d/%04d",d.da_day,d.da_mon,
        d.da_year);
 printf("\tora\
   %02d:%02d:%02d\n",t.ti_hour,t.ti_min,t.ti_sec);
}
```

17.2 Să se scrie un program care afişează un şir de numere naturale pseudo-aleatoare mai mici decît 10*(sec+1), unde prin *sec* s-a notat secunda din ora curentă.

PROGRAMUL BXVII2

```
#include <dos.h>
#include <stdlib.h>
#include <stdio.h>

main() /* - afiseaza 3 numere pseudo-aleatoare mai mici decit
           10*(sec+1);
          - sec este secunda curenta. */
{
 struct time ora_crt;
 unsigned sec;
 int i;

 gettime(&ora_crt);
 sec=ora_crt.ti_sec;
 sec++;
 for(i=0;i<3;i++)
    printf("%d\n",random(10*sec));
}
```

17.3 Să se scrie un program care afişează un şir de numere naturale pseudo-aleatoare mai mici decît *n*, citit de la intrarea standard. Primul număr se afişează automat, iar celelalte în urma acţionării unei taste.

Sămînţa şirului de numere pseudo-aleatoare *s*, se calculează astfel:

s=(ti_min*60 + ti_hour*3600 + ti_sec) % 65535;

Programul se întrerupe la tastarea cifrei zero.

PROGRAMUL BXVII3

```
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <conio.h>

main () /* afiseaza numere pseudo-aleatoare din intervalul [0,n] */
{
 struct time ora_crt;
 int i,n,s;

 for(;;){
    printf("n=");
```

```
        if((i=scanf("%d",&n))==1&&n>0)
            break;
        printf("nu s-a tastat un intreg pozitiv\n");
        if(i==EOF){
            printf("s-a tastat EOF \n");
            exit(1);
        }

        fflush(stdin); /* videaza zona tampon a fisierului de
                          intrare standard */
    }
    gettime(&ora_crt);
    s=(3600L*ora_crt.ti_hour+
             60*ora_crt.ti_min+ora_crt.ti_sec)%65535;
    srand(s);
    for(;;){
        printf("%d\n",random(n));
        printf("Pentru a continua actionati o tasta\
                diferita de zero\n");
        if(getch()=='0')
            exit(0);
    }
}
```

Observaţie:

Numerele pseudo-aleatoare obţinute în acest fel se pot utiliza în diferite aplicaţii pentru a simula situaţii reale. Dacă n=6, atunci numerele pseudo-aleatoare rezultate, mărite cu 1, pot simula aruncarea unui zar.

17.4 Să se scrie un program care citeşte numere de la intrarea standard şi le copiază într-un fişier binar pe disc. Calea spre fişier este definită de argumentul programului. În caz de eroare, programul se termină apelînd funcţia *exit* cu următoarele coduri de stare:

1 - Nu se poate deschide fişierul definit de argumentul programului.

2 - Eroare la scrierea numerelor în fişier.

3 - Eroare la închiderea fişierului.

4 - Fişier vid.

La terminarea programului fără erori se apelează funcţia *exit* cu codul de stare zero.

PROGRAMUL BXVII4

```
#include <stdio.h>
#include <stdlib.h>

main(int argc,char *argv[])
/* citeste numere de la intrarea standard si le pastreaza in fisierul a carui
   cale este definita de argumentul argv[1] */
{
 FILE *pf;
 int i,n;
 double f[10];
 void *zt;
 double x;

 if(argc!=2){
     printf("nr arg=%d\n" ,argc);
     printf("%s:numar argumente eronat\n",argv[0]);
     exit(1);
 }
 if((pf=fopen(argv[1],"wb"))==0){
     printf("%s:nu se poate deschide\
           fisierul:%s\n",
     argv[0],argv[1]);
     exit(1);
 }
 n=0;
 zt=(void *)f;
 printf("Tastati numerele separate prin caractere\
       albe\n");
 for(;;){
     for(i=0;i<10;i++){
         if(scanf("%lf",&x)!=1)
             break;
         f[i]=x;
     }
     if(i==0)
         break;
     if(fwrite(zt,sizeof(double),i,pf)!=i){
         printf("%s:eroare la scrierea in\
               fisierul:%s\n",
         argv[0],argv[1]);
         exit(2);
     }
```

```
        n++;
        if(i<10)
            break;
    }
    if(n==0){
        printf("%s:nu s-a citit nici un\
            numar\n",argv[0]);
        exit(4);
    }
    if(fclose(pf)==0)
        exit(0);
    printf("%s:eroare la inchiderea fisierului:%s\n",
        argv[0],argv[1]);
    exit(3);
}
```

17.5 Să se scrie un program care realizează următoarele:

a. Citeşte numere de la intrarea standard şi le păstrează în fişierul *dnr.dat*.

b. Folosind utilitarul *pkzip.exe*, determină forma condensată *dnr.zip* a fişierului *dnr.dat*.

c. Şterge fişierul *dnr.dat*.

Punctul *a* se realizează apelînd programul din exerciţiul precedent. În acest scop vom folosi funcţia *spawnl*.

Punctul *b* se realizează apelînd utilitarul *pkzip* folosind funcţia *system*.

La punctul *c* se apelează funcţia *unlink*.

Utilitarul *pkzip.exe* se află în directorul *arc* de pe discul c. Se va utiliza linia de comandă:

c:\arc\pkzip.exe -a dnr.zip dnr.dat

PROGRAMUL BXVII5

```
#include <stdio.h>
#include <process.h>

main() /* citeste numere de la intrarea standard si le pastreaza
          in fisierul dnr.zip */
{
 char *text[]={
    "",
```

```
        "cale spre fisier eronata\n",
        "eroare la scriere in fisier\n",
        "nu se poate inchide fisierul\n",
        "fisier vid\n"
    };
    int cod;

    if((cod=
      spawnl(P_WAIT,"C:\\borlandc\\dest\\BXVII4.EXE",
              "","dnr.dat",0))!=0){
        printf("\n\n***%s\n",text[cod]);
        exit(1);
    }
    unlink("dnr.zip");
    if(system("c:\\arc\\pkzip.exe -a\
            dnr.zip dnr.dat")!=0){
        printf("eroare la apelul utilitarului\
              pkzip\n");
        exit(1);
    }
    if (unlink("dnr.dat")!=0){
        printf("eroare la stergerea fisierului\
              dnr.dat\n");
        exit(1);
    }
}
```

Observaţie:

Utilitarul *pkzip* afişează o serie de informaţii la terminalul standard care uneori poate afecta aspectul ecranului cu datele aplicaţiei. Pentru a elimina acest inconvenient, se poate face o redirectare pentru a afişa datele respective pe un alt periferic. În acest caz funcţia *system* se va apela folosind şirul de caractere:

"c:\\arc\\pkzip.exe -a dnr.zip dnr.dat >nume"

În acest caz, informaţiile afişate de *pkzip* se vor scrie în fişierul *nume* din directorul curent. Aceste informaţii pot fi suprimate dacă redirectarea se face spre perifericul NUL, adică înlocuind *nume* din şirul de caractere de mai sus cu NUL.

17.6 Să se scrie un program care citeşte de la intrarea standard un text şi-l redă prin sunet utilizînd alfabetul Morse.

Se redau numai literele din compunerea cuvintelor. Restul caracterelor se neglijează.

Pentru redarea *punctului* se utilizează un semnal cu o frecvenţă de 1000 de Hz pe o durată de 0,075 secunde.

Pentru redarea liniei se utilizează acelaşi semnal, dar pe o durată de trei ori mai mare.

De asemenea, între două caractere din compunerea unei litere se face pauză de 0,075 secunde. Între două litere din compunerea aceluiaşi cuvînt se face o pauză de 0,225 secunde, iar între două cuvinte o pauză de 0,375 secunde.

O codificare similară a unui text este propusă în exerciţiul P.131 din Gazeta de Informatică nr. 7 - 8/1992 (pag 48). În acest caz textul se codifică printr-o secvenţă de cifre 0 şi 1.

PROGRAMUL BXVII6

```
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <string.h>

#include "bx48.cpp"

#define MAX 1000
#define FRECV 1000
#define UNIT 75
#define INTERCAR 225
#define INTERCUV 375

main() /* citeste un text de la intrarea standard si-l reda prin sunet
          utilizind alfabetul Morse */
{
 char *pcuv[MAX];
 int c,i,n;
 static char *morse[]={
     ".-","-...","-.-.","-..",".","..-.",
     "--.","....", "..",".---","-.-",".-..",
     "--","-.","---",".--.",   "--.-",".-.",
     "...","-","..-","...-",".--","-..-","-.--",
     "--.."
  };
```

```
 char *p,*q;

/* se citesc cuvintele din text si se pastreaza in memoria heap */
 for(n=0;n<MAX;n++)
     if((pcuv[n]=citcuv())==0)
         break; /* s-a intilnit EOF */
 if(n==0){
     printf("nu s-a citit nici un cuvint\n");
     exit(1);
 }

/* redarea textului prin sunet conform alfabetului Morse */
 for(i=0;i<n;i++){
     p=pcuv[i];
     while(*p){
         if((c=*p++)>='a')
             c=c-'a'; /* litera mica */
         else
             c=c-'A'; /* litera mare */
         for(q=morse[c];*q;){
             sound(FRECV);
             if(*q=='.')
                 delay(UNIT); /* punct */
             else
                 delay(3*UNIT); /* linie */
             nosound();
             if(*++q) /* nu s-a terminat litera curenta */
                 delay(UNIT);
         }
         if(*p) /* s-a terminat o litera */
             delay(INTERCAR);
     }
 /* s-a terminat un cuvint */
 delay(INTERCUV);
 }
}
```

18. GESTIUNEA ECRANULUI ÎN MOD TEXT

Biblioteca standard a limbajului C şi C++ conţine funcţii pentru gestiunea ecranului. Acesta poate fi gestionat în două moduri:

mod text;

sau

mod grafic.

În capitolul de faţă prezentăm funcţiile standard mai importante utilizate la gestiunea ecranului în mod text.

În capitolul următor se tratează gestiunea ecranului în mod grafic.

Toate funcţiile standard de gestiune a ecranului în mod text au prototipurile în fişierul *conio.h*.

Modul *text* presupune că ecranul este format dintr-un număr de linii şi un număr de coloane. În mod curent se utilizează 25 de linii a 80 sau 40 de coloane fiecare. Aceasta înseamnă că ecranul are o capacitate de 25*80=2000 sau 25*40=1000 caractere.

Poziţia pe ecran a unui caracter se defineşte printr-un sistem de două coordonate întregi:

(x,y)

unde:

x - Este numărul coloanei în care este situat caracterul.

y - Este numărul liniei în care este situat caracterul.

Colţul din stînga sus al ecranului are coordonatele (1,1). Colţul din dreapta jos al ecranului are coordonatele (80,25) sau (40,25).

În mod implicit, funcţiile de gestiune a ecranului în mod text au acces la tot ecranul. Accesul poate fi limitat la o parte din ecran utilizînd aşa numitele *ferestre*. *Fereastra* este un dreptunghi care este o parte a ecranului şi care poate fi gestionată independent de restul ecranului.

Un caracter de pe ecran, pe lîngă coordonate, mai are şi următoarele atribute:

- culoarea caracterului afişat;
- culoarea fondului;

– clipirea caracterului.

Aceste atribute sînt dependente de adaptorul grafic utilizat. Cele mai utilizate adaptoare sînt:

- placa MDA, care este un adaptor monocrom;
- placa Hercules, care este un adaptor monocrom;
- placa CGA, care este un adaptor color;
- placa EGA, care este un adaptor color;
- placa VGA, care este un adaptor color de mare performanţă.

Pentru adaptoarele color de mai sus, se pot utiliza 8 culori de fond şi 16 pentru afişarea caracterelor.

Atributul unui caracter se defineşte cu ajutorul formulei:

(1) atribut = 16 * culoare_fond +culoare_caracter + clipire.

unde:

culoare_fond (background)	- Este o cifră din intervalul [0,7] şi are semnificaţia din tabela de mai jos.
culoare_caracter (foreground)	- Este un întreg din intervalul [0,15] şi are semnificaţia din tabela de mai jos.
clipire	- Are valoarea 128 (clipirea caracterului) sau 0 (fără clipire).

În tabelul de mai jos se indică corespondenţa dintre valorile numerice şi culorile definite de ele cu ajutorul relaţiei (1).

Culoare	**Constantă simbolică**	**Valoare**
negru	BLACK	0
albastru	BLUE	1
verde	GREEN	2
turcoaz	CYAN	3
roşu	RED	4
purpuriu	MAGENTA	5
maro	BROWN	6
gri deschis	LIGHTGRAY	7
gri închis	DARKGRAY	8
albastru deschis	LIGHTBLUE	9
verde deschis	LIGHTGREEN	10
turcoaz deschis	LIGHTCYAN	11
roşu deschis	LIGHTRED	12

continuare

Culoare	Constantă simbolică	Valoare
purpuriu deschis	LIGHTMAGENTA	13
galben	YELLOW	14
alb	WHITE	15
clipire	BLINK	128

În paragrafele următoare se indică funcţiile standard mai importante pentru gestiunea ecranului în mod text.

18.1. Setarea ecranului în mod text

Se realizează cu ajutorul funcţiei *textmode*. Aceasta are prototipul:

void textmode(int *modtext***);**

unde:

modtext - Poate fi exprimat numeric sau simbolic în felul următor:

Modul text activat	Constantă simbolică	Valoare
Caractere albe pe fond negru; 40 de coloane	BW40	0
Color 40 de coloane	C40	1
Caractere albe pe fond negru; 80 de coloane	BW80	2
Color 80 de coloane	C80	3
Monocrom 80 de coloane	MONO	7
Color cu 43 linii pentru placa EGA şi 50 de linii pentru placa VGA	C4350	64
Modul precedent	LASTMODE	-1

Modul MONO se poate seta pe un adaptor monocolor.

Celelalte moduri se pot seta pe adaptoare color.

18.2. Definirea unei ferestre

După setarea ecranului în mod text, acesta are caracteristicile indicate în paragraful precedent.

Adesea dorim să partajăm ecranul în zone care să poată fi gestionate independent. Aceasta se realizează cu ajutorul ferestrelor.

O *fereastră* este o zonă dreptunghiulară de pe ecran. Ea se poate defini cu ajutorul funcţiei *window*. Prototipul ei este:

void window(int *stinga*,**int** *sus*, **int** *dreapta*,**int** *jos*);

unde:

(stinga, sus) - Reprezintă coordonatele colţului din stînga sus al ferestrei.

(dreapta,jos) - Reprezintă coordonatele colţului dreapta jos al ferestrei.

La un moment dat o singură fereastră este *activă* şi anume aceea definită de ultimul apel al funcţiei *window*.

Funcţiile de gestiune a ecranului în mod text acţionează totdeauna asupra ferestrei active.

După setarea modului text cu ajutorul funcţiei *textmode*, este activ tot ecranul.

Menţionăm că funcţia *window* nu are nici un efect dacă parametri de la apel sînt eronaţi.

18.3. Ştergerea unei ferestre

Fereastra activă se şterge cu ajutorul funcţiei *clrscr*. Ea are prototipul:

void clrscr(void);

După apelul funcţiei *clrscr*, fereastra activă (sau tot ecranul, dacă nu s-a definit în prealabil o fereastră prin apelul funcţiei *window*) devine vidă. Fondul ei are culoarea definită prin culoarea de fond (*background*) curentă.

Funcţia *clrscr* poziţionează *cursorul* pe caracterul din stînga sus al ferestrei active, adică în poziţia de coordonate (1,1) a ferestrei active.

18.4. Gestiunea cursorului

Utilizatorul poate plasa cursorul pe un caracter al ferestrei folosind

funcţia *gotoxy*. Ea are prototipul:

void gotoxy(int *coloana*,**int** *linie*);

unde:

(coloana,linie) - Reprezintă coordonatele caracterului pe care se plasează cursorul; aceste coordonate sînt relative la fereastra activă.

Dacă coordonatele de la apel sînt în afara ferestrei active, atunci apelul funcţiei este ignorat.

Poziţia cursorului din fereastra activă se poate determina cu ajutorul a două funcţii, care au prototipurile:

int wherex(void);

returnează numărul coloanei în care se află cursorul;

int wherey(void);

returnează numărul liniei în care se află cursorul.

Există cazuri cînd se doreşte *ascunderea* cursorului. Acest lucru se poate realiza printr-o secvenţă specială în care se utilizează funcţia *geninterrupt*. O secvenţă de acest fel este următoarea:

```
void ascundecursor() /* face invizibil cursorul */
{
 _AH = 1;
 _CH = 0x20;
 geninterrupt(0x10);
}
```

Cursorul poate fi reafişat apelînd funcţia de mai jos:

```
void afiscursor() /* face vizibil cursorul */
{
 _AH = 1;
 _CH = 6;
 _CL = 7;
 geninterrupt(0x10);
}
```

Amintim că _AH, _CH şi _CL sînt nume utilizate pentru regiştrii calculatorului.

18.5. Determinarea parametrilor ecranului

Utilizatorul are posibilitatea să obţină parametri curenţi ai ecranului prin apelarea funcţiei *gettextinfo*. Ea are prototipul:

void gettextinfo(struct text_info *p);

unde structura *text_info* este definită în fişierul *conio.h* astfel:

```
struct text_info {
        unsigned char winleft;
        unsigned char wintop;
        unsigned char winright;
        unsigned char winbottom;
        unsigned char attribute;
        unsigned char normattr;
        unsigned char currmode;
        unsigned char screenheight;
        unsigned char screenwidth;
        unsigned char currx;
        unsigned char curry;
};
```

După apelul funcţiei *gettextinfo* structura de tip *text_info*, spre care pointează *p*, este completată cu următoarele informaţii:

- amplasarea colţurilor ferestrei;
- culoarea fondului, a caracterelor şi clipirea acestora;
- modul curent;
- dimensiunea ecranului;
- poziţia cursorului.

18.6. Modurile video alb/negru

Modurile video alb/negru sînt două:

modul intens

şi

modul normal.

Modul intens se obţine apelînd funcţia *highvideo* de prototip:

void highvideo(void);

Modul normal se obţine cu ajutorul funcţiei *lowvideo* de prototip:

void lowvideo(void);

Intensitatea iniţială este de obicei cea normală. Se poate reveni la intensitatea normală dacă se apelează funcţia *normvideo* de prototip:

void normvideo(void);

18.7. Setarea culorilor

Culoarea fondului se setează cu ajutorul funcţiei *textbackground* de prototip:

void textbackground(int *culoare***);**

unde:

culoare - Este un întreg în intervalul [0,7] şi are semnificaţia definită în tabelul de la începutul acestui capitol.

Culoarea caracterelor se setează cu ajutorul funcţiei *textcolor* de prototip:

void textcolor(int *culoare***);**

unde:

culoare - Este un întreg din intervalul [0,15] şi are semnificaţia definită în tabelul de la începutul acestui capitol.

Se pot seta ambele culori, precum şi clipirea caracterului folosind funcţia *textattr* de prototip:

void textattr(int *atribut***);**

unde:

atribut - Se defineşte cu ajutorul relaţiei (1).

Exerciţii:

18.1 Să se scrie o funcţie care afişează parametrii ecranului.

FUNCŢIA BXVIII1

```
void pecr() /* afiseaza parametrii ecranului */
{
 struct text_info parecr;

 clrscr();
```

```
 gettextinfo(&parecr);
 printf("stinga:%u sus:%u dreapta:%u jos:%u\n",
      parecr.winleft,parecr.wintop,
      parecr.winright, parecr.winbottom);
 printf("atribut:%u mod curent:%u\n",
      parecr.attribute,parecr.currmode);
 printf("inaltimea ecranului:%u latimea\
      ecranului:%u\n",parecr.screenheight,
       parecr.screenwidth);
 printf("coloana cursorului:%u linia\
      cursorului:%u\n",parecr.curx,parecr.cury);
}
```

18.2 Să se scrie un program care setează pe rînd modurile text, definite cu ajutorul constantelor simbolice:

BW40, C40, BW80, C80 şi C4350

şi afişează parametrii ecranului pentru fiecare din modurile respective.

PROGRAMUL BXVIII2

```
#include <stdio.h>
#include <conio.h>

#include "bxviii1.cpp"

main() /* seteaza modurile definite cu ajutorul constantelor simbolice
          BW40, C40, BW80, C80 si C4350 si afiseaza parametrii
          ecranului in fiecare caz */
{
 int i;
 int tab[]={BW40,C40,BW80,C80,C4350};
 char *text[]={
    "BW40","C40","BW80","C80","C4350"
 };

 for(i=0;i<5;i++){
    textmode(tab[i]);
    pecr();
    printf("\n\t\t\t%s\n\n",text[i]);
    printf("Actionati o tasta pentru a\
          continua\n");
    getch();
```

```
    }
  }
```

Observaţie:

Programul de faţă presupune prezenţa la calculator a unui adaptor color de tip EGA sau VEGA.

18.8. Gestiunea textelor

Pentru afişarea caracterelor colorate în conformitate cu atributele definite prin relaţia:

atribut = 16*culoare_fond + culoare_caracter + clipire

se pot folosi funcţiile:

putch - Afişează un caracter.

cputs - Afişează un şir de caractere (este analogă cu funcţia *puts*).

cprintf - Afişează date sub controlul formatelor de conversie (este analogă cu funcţia *printf*).

Toate aceste funcţii au prototipul în fişierul *conio.h*.

Atributul de culoare şi clipire se setează cu ajutorul funcţiilor indicate în paragraful 18.7.

Biblioteca standard a limbajului C conţine şi alte funcţii utile în gestiunea textelor. Dintre acestea amintim pe cele mai importante.

Operaţiile de ştergere şi inserare de linie se pot realiza prin funcţiile de mai jos:

void insline(void);

inserează o linie cu spaţii în fereastră; liniile de sub poziţia curentă a cursorului se deplasează în jos cu o poziţie;

void clreol(void);

şterge sfîrşitul liniei începînd cu poziţia cursorului;

void delline(void);

şterge toată linia pe care este poziţionat cursorul.

Un text poate fi copiat dintr-o zonă dreptunghiulară a ecranului în alta, folosind funcţia *movetext*. Ea are prototipul:

int movetext(int *stanga*, **int** *sus*, **int** *dreapta*, **int** *jos*,
int *stanga_dest*, **int** *sus_dest*);

unde:

stanga,sus	- Definesc începutul textului care se copiază.
dreapta,jos	- Definesc sfîrşitul textului care se copiază.
stanga_dest, sus_dest	- Definesc poziţia primului caracter al textului după copiere. Celelalte poziţii rezultă automat din structura textului.

Funcţia returnează valoarea:

1	- Dacă textul s-a copiat corect.
0	- La eroare.

Textele dintr-o zonă dreptunghiulară pot fi salvate într-o zonă de memorie sau citite dintr-o astfel de zonă folosind funcţiile *gettext* şi *puttext*. Ele au prototipurile de mai jos:

int gettext(int *stanga*, **int** *sus*, **int** *dreapta*, **int** *jos*, **void** **destinatie*);

unde:

stanga,sus dreapta,jos	- Definesc o zonă dreptunghiulară din ecran care conţine textul de salvat.
destinatie	- Este pointerul spre zona de memorie în care se salvează textul.

Funcţia returnează:

1	- La copiere cu succes.
0	- La eroare.

int puttext(int *stanga*, **int** *sus*, **int** *dreapta*, **int** *jos*, **void** **sursa*);

unde:

stanga,sus dreapta,jos	- Definesc o zonă dreptunghiulară din ecran în care se va afişa textul citit din memorie.
sursa	- Este pointerul spre zona de memorie din care se transferă textul pe ecran.

Funcţia returnează:

1	- La copiere cu succes;
0	- La eroare.

Menţionăm că fiecare caracter de pe ecran se păstrează pe doi octeţi:

- pe un octet caracterul;
- pe octetul următor atributul caracterului.

Exerciţii:

18.3 Să se scrie un program care afişează texte în modurile video intens şi video normal.

PROGRAMUL BXVIII3

```
#include <conio.h>

main() /* afiseaza texte in modurile video intens si normal */
{
 textmode(BW80);
 window(10,4,60,4);
 clrscr();
 lowvideo();
 cputs("lowvideo");
 highvideo();
 cputs(" highvideo");
 normvideo();
 cputs(" normvideo");
 textmode(LASTMODE);
 cprintf("\n\rActionati o tasta pentru a\
        continua" );
 getch();
}
```

18.4 Să se scrie un program care afişează toate combinaţiile de culori posibile pentru fond şi caractere. Se consideră că se dispune de un adaptor color EGA/VGA.

PROGRAMUL BXVIII4

```
#include <conio.h>
#include <stdio.h>

main() /* - afiseaza toate combinatiile de culori posibile pentru
            fond si caractere;
          - se dispune de un adaptor EGA/VGA. */
{
 static char *tculoare[] = {
    "  0  BLACK                   negru",
```

```
   "    1   BLUE                       albastru",
   "    2   GREEN                      verde",
   "    3   CYAN                       turcoaz",
   "    4   RED                        rosu",
   "    5   MAGENTA                    purpuriu",
   "    6   BROWN                      maro",
   "    7   LIGHTGRAY                  gri deschis",
   "    8   DARKGRAY                   gri inchis",
   "    9   LIGHTBLUE                  albastru deschis",
   "    10  LIGHTGREEN                 verde deschis",
   "    11  LIGHTCYAN                  turcoaz deschis",
   "    12  LIGHTRED                   rosu deschis",
   "    13  LIGHTMAGENTA               purpuriu deschis",
   "    14  YELLOW                     galben",
   "    15  WHITE                      alb"
};

int i,j,k;
struct text_info atr;

gettextinfo(&atr);
for(i=0; i <8; i++){

/* i alege culoarea fondului */
    window(3,2,60,20);
    k=2;
    textbackground(i);
    clrscr();
    for(j=0; j < 16; j++, k++) {
        textcolor(j);
        gotoxy(2,k);

/* j alege culoarea caracterului */
        if(i == j)
            continue;
        cputs(tculoare[j]);
    }
    gotoxy(1,18);
    printf("actionati o tasta pentru a\
          continua\n");
    getch();
}
window(atr.winleft,atr.wintop,
```

```
        atr.winright,atr.winbottom);
    textattr(atr.attribute);
    clrscr();
}
```

18.5 Să se scrie un program pentru rezolvarea problemei turnurilor din Hanoi.

Această problemă a fost rezolvată în exercițiul 9.6. Mai jos se rezolvă aceeaşi problemă prin imagini pentru n < 7, *n* fiind numărul de discuri (n > 0). Se presupune că se dispune de un adaptor color EGA/VGA.

Programul foloseşte o matrice de ordinul 3*6 pentru a păstra discurile pe cele trei tije. Numim *stiva* tabloul care păstrează elementele acestei matrice. Ea este de tip *int*. Amintim că discul de cel mai mic diametru se numerotează cu 1, discul cu diametru imediat mai mare decît acesta se numerotează cu 2 şi aşa mai departe. Dacă sînt *n* discuri, atunci discul cu cel mai mare diametru se numerotează cu *n*.

Discurile aflate pe tija A se păstrează ca elemente ale tabloului *stiva* pentru care primul indice este zero:

stiva[0][0], stiva[0][1], stiva[0][2], ...

Discurile aflate pe tija B se păstrează ca elemente ale aceluiaşi tablou pentru care primul indice are valoarea unu:

stiva[1][0], stiva[1][1], stiva[1][2], ...

În mod analog, pentru discurile aflate pe tija C se utilizează elementele pentru care primul indice are valoarea doi:

stiva[2][0], stiva[2][1], stiva[2][2], ...

Inițial discurile sînt pe tija A, deci

stiva[0][i] = n-i pentru i = 0,1,...,n-1.

Un alt tablou utilizat în program determină numărul discurilor aflate pe fiecare tijă. Numim *istiva* acest tablou. El are 3 elemente: istiva[0] are ca valoare numărul discurilor aflate pe tija A, istiva [1] are ca valoare numărul discurilor de pe tija B, iar istiva[2] are ca valoare numărul discurilor de pe tija C.

Inițial istiva[0] = n, istiva[1] = 0 şi istiva[2] = 0 deoarece toate cele *n* discuri se află pe tija A.

Cele trei tije se reprezintă prin trei dreptunghiuri albastre. Primul are coordonatele:

- colţul din stînga sus (7,3);
- colţul din dreapta jos (8,17).

Următorul se obţine făcînd o translaţie cu 28 de coloane spre dreapta; deci acesta are coordonatele (7+28,3) şi (8+28,17).

Ultimul dreptunghi se obţine printr-o nouă translaţie tot de 28 de coloane.

Sub fiecare dreptunghi se afişează cu galben litera tijei pe care o reprezintă (A, B sau C).

Discurile se reprezintă prin dreptunghiuri colorate şi acestea au culorile:

discul	n	LIGHTGRAY
	n-1	BROWN
	n-2	MAGENTA
	n-3	RED
	n-4	CYAN
	n-5	GREEN

Fondul ecranului este negru.

Mutările discurilor se realizează cu ajutorul funcţiei recursive *hanoi*. Aceasta este analogă cu cea definită în exerciţiul 9.6. În cazul de faţă, pentru n < 7, în loc să se afişeze mutarea sub forma unui text, se realizează deplasarea prin imagini a discului care se mută de pe o tijă pe alta. Acest lucru se realizează apelînd funcţia *transfer*. Aceasta are prototipul:

void *transfer*(**int** *tija1*, **int** *tija2*);

Ea transferă discul din vîrful tijei *tija1*, în vîrful tijei *tija2*.

PROGRAMUL BXVIII5

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

int stiva[3][6];
int istiva[3];
int m;

typedef struct {
 int x;
 int y;
```

```
 int u;
 int v;
 int poz;
}F;

struct text_info parecr;

void inithanoi (int n) /* initializeaza discurile pe tija A */
{
 int x,y,u,v,i,lit,culoare;

/* salveaza parametrii ecranului */
 gettextinfo (&parecr);

/* fond negru pentru tot ecranul */
 textbackground (BLACK);

/* culoarea caracterelor se seteaza la galben */
 textcolor (YELLOW);

/* sterge ecranul */
 clrscr ();

/* se amplaseaza cele trei tije de culoare albastra fiecare */
 x = 7;
 y = 3;
 u = 8;
 v = 17;
 lit= 'A';
 for (i = 0; i<3; i++){
     window(x,y,u,v);
     textbackground(BLUE);
     clrscr();

/* fereastra pentru litera */
     window (x,v,u,v+2);

/* fond negru si caracter galben = 14 */
     textattr (14);
     clrscr ();

/* scrie litera */
     putch (lit);
```

```
/* se modifica parametrii pentru tija urmatoare */
    lit++;
    x = x+28;
    u = u+28;
 }

/* se pun discurile pe tija A */
/* culorile discurilor au valorile 7,6,...,2 */
/* discul n are lungimea 2*n+1 caractere si inaltimea de 1 caracter */
 culoare = 7;
 x = 7-n;
 y = 15;
 u = 8+n;
 for (i = n; i>0; i--){
     window (x,y,u,v-1);
     textbackground (culoare);
     clrscr ();
     culoare--;
     x++;
     u--;
     v = y;
     y = y-2;
 }

/* se initializeaza tablourile stiva si istiva */
 for (i = 0; i<n; i++)
     stiva [0][i] = n-i;
 istiva[0] = i;
 istiva[1] = 0;
 istiva[2] = 0;
 getch(); /* afiseaza starea initiala */
} /* sfirsit inithanoi */

void fereastra (F *frstr,int nrstv,
                int vs,int ndisc)
/* defineste fereastra activa pe discul ndisc aflat in virful vs pe tija
   nrstv */
{
 int poz;

/* se determina pozitia tijei */
 poz = (nrstv*4 + 1)*7; /* nrstv = 0 tija A;
                                  1 tija B;
```

```
                        2 tija C;
                   poz - pozitia tijei. */

 /* se determina coordonatele discului ndisc */
  frstr -> x = poz - ndisc;
  frstr -> u = poz+1+ndisc;
  frstr -> y = 17 -2*vs; /* vs - numar discuri pe tija nrstv */
  frstr -> v = frstr -> y +1;

 /* se activeaza fereastra pe discul ndisc */
  window (frstr -> x, frstr -> y, frstr -> u,
          frstr -> v);
  frstr -> poz = poz; /* pastreaza pozitia tijei */
 } /* sfirsit fereastra */

void transfer ( int tija1,int tija2)
/* deplaseaza discul din virful tijei tija1 in virful tijei tija2 */
{
 int i,j,k,l,nr;
 F cf;

/* determina numarul tijei tija1 */
/* tija A are numarul 0, tija B are numarul 1, tija C are numarul 2 */
 if (tija1 == 'A')
     i = 0;
 else
     if (tija1 == 'B')
         i = 1;
     else
         i = 2;

/* i - numarul tijei tija1;
   j - numarul discurilor pe tija tija1;
  nr - numarul discului din virful tijei tija1. */
 j = istiva[i];
 nr = stiva[i][j-1];

/* - determina fereastra cu discul din virful tijei a i-a;
   - aceasta devine activa. */
 fereastra (&cf,i,j,nr);

/* sterge fereastra din virful stivei i */
 textbackground (BLACK);
```

```
clrscr ();

/* reface tija a i-a */
window ( cf.poz, cf.y, cf.poz+1, cf.v);
textbackground (BLUE);
clrscr ();

/* - se scoate discul din stiva a i-a;
  - acesta corespunde discului care a fost in virful tijei tija1 */
istiva[i]--;

/* determina numarul tijei tija2 */
if (tija2 == 'A')
    i=0;
else
    if (tija2 == 'B')
        i = 1;
    else
        i = 2;

/* pune discul nr pe tija a i-a */
stiva[i] [istiva[i]] = nr;
istiva[i]++;

/* determina fereastra pe tija a i-a pentru discul nr */
fereastra (&cf,i,istiva[i],nr);

/* fereastra activa devine fereastra de pe tija tija2 in virful careia se va
  pune discul nr */
/* stabileste culoarea ferestrei */
textbackground(7-m+nr);
clrscr ();
} /* sfirsit transfer */

void hanoi(int n,int a,int b,int c)
/* functia recursiva pentru rezolvarea problemei turnurilor din Hanoi */
{
 if (n == 1)
     if (m>6){
         printf("se muta discul 1 de pe tija: %c pe\
                tija: %c\n",a,b);
         getch();
         return;
```

```
        }
        else{
            transfer(a,b);
            getch();
            return;
        }
    hanoi(n-1,a,c,b);
    if(m > 6){
        printf("discul: %d se muta de pe tija :%c pe\
              tija: %c\n",n,a,b);
        getch();
    }
    else{
        transfer (a,b);
        getch();
    }
    hanoi(n-1,c,b,a);
} /* sfirsit hanoi */

main () /* rezolva problema turnurilor din Hanoi */
{
 int i,c;
 char t[255];

 for(;;){
     printf("numarul discurilor=");
     if(gets(t)==0){
         printf("s-a tastat EOF\n");
         exit(1);
     }
     if(sscanf(t,"%d",&m)==1&&m>0)
         break;
     printf("nu s-a tastat un intreg pozitiv\n");
 }
 if(m < 7){
     printf("rezolvare insotita de imagini\n");
     inithanoi(m);
 }
 else
     printf("rezolvarea nu este insotita de\
           imagini\n");
 hanoi (m,'A','B','C');
 if(m < 7){
```

```
        window (1,1,parecr.screenwidth,
                parecr.screenheight);
        textattr (parecr.attribute);
        clrscr();
    }
} /* sfirsit main */
```

18.6 Să se scrie o funcţie care afişează o fereastră limitată de un chenar şi pe fondul căreia se afişează un întreg. Cursorul devine invizibil la afişarea ferestrei.

Funcţia are prototipul:

void *fereastra*(**int** *st*, **int** *sus*, **int** *dr*, **int** *jos*, **int** *fond*,
int *culoare*, **int** *chenar*, **int** *n*);

unde:

(st,sus) - Coordonatele colţului din stînga sus.

(dr,jos) - Coordonatele colţului din dreapta jos.

fond - Întreg din intervalul [0,7] care defineşte culoarea de fond a ferestrei.

culoare - Întreg din intervalul [0,15] care defineşte culoarea pentru afişarea caracterelor.

chenar - Defineşte tipul chenarului:

- 0 - fără bordură;
- 1 - linie simplă;
- 2 - linie dublă;
- n -numărul întreg care se afişează în fereastră începînd cu punctul de coordonate relative (3,3).

Zona de ecran în care se afişează fereastra se păstrează în memoria *heap* înainte de a se afişa fereastra. Adresa acestei zone de memorie se pune pe o stivă definită cu ajutorul unui tablou de pointeri spre tipul *void*. Acest tablou este global şi are 100 de elemente. El se defineşte astfel:

void far *stiva[100];

Locul liber în tablou se defineşte cu ajutorul variabilei globale *istiva*:

int istiva;

FUNCŢIA BXVIII6

```
void orizontal(int,int);
```

```
void vertical(int,int,int,int);

void fereastra(int st,int sus,int dr,int jos,
               int fond,int culoare,int chenar,int n)
```

/* afiseaza o fereastra limitata de un chenar si pe fondul careia se afiseaza numarul ferestrei */

```
{
 extern ELEM far *stiva[];
 extern int istiva;
```

/* memoreaza partea din ecran pe care se va afisa fereastra */

```
 if(istiva==MAX){
     printf("\nprea multe ferestre\n");
     exit(1);
 }
 if((stiva[istiva]=
           (ELEM *)farmalloc(sizeof(ELEM)))==0){
     printf("memorie insuficienta\n");
     exit(1);
 }
 if((stiva[istiva]->zonfer=
        farmalloc(2*(dr-st+1)*(jos-sus+1))))==0){
     printf("\nmemorie insuficienta\n");
     exit(1);
 }
 stiva[istiva]->x=st;
 stiva[istiva]->y=sus;
 stiva[istiva]->u=dr;
 stiva[istiva]->v=jos;
 if((gettext(st,sus,dr,jos,stiva[istiva]->
        zonfer))==0){
     printf("\neroare la memorarea ecranului\n");
     exit(1);
 }
 istiva++;
```

/* activeaza fereastra si o afiseaza pe ecran */

```
 window(st,sus,dr,jos);
 textattr(16*fond+culoare);
 clrscr();
```

/* trasare chenar */

```
  if(chenar){
      textcolor(WHITE);
      highvideo();

 /* coltul stinga sus */
      switch(chenar){
          case SIMPLU:
              putch(218);
              break;
          case DUBLU:
              putch(201);
              break;
      }

 /* chenar orizontal sus */
      orizontal(dr-st-2,chenar);

 /* coltul dreapta sus */
      switch(chenar){
          case SIMPLU:
              putch(191);
              break;
          case DUBLU:
              putch(187);
              break;
      }

 /* chenar vertical stinga */
      vertical(jos-sus,1,2,chenar);

 /* coltul stinga jos */
      gotoxy(1,jos-sus+1);
      switch(chenar){
          case SIMPLU:
              putch(192);
              break;
          case DUBLU:
              putch(200);
              break;
      }

 /* chenar orizontal jos */
      orizontal(dr-st-2,chenar);
```

```
/* chenar vertical dreapta */
     vertical(jos-sus-1,dr-st,2,chenar);

/* coltul dreapta jos */
     gotoxy(dr-st,jos-sus+1);
     switch(chenar){
         case SIMPLU:
             putch(217);
             break;
         case DUBLU:
             putch(188);
             break;
     }
     normvideo();
     textattr(16*fond+culoare);
  } /* sfirsit afisare chenar */

/* scrie pe n in fereastra */
 gotoxy(3,3);
 cprintf("%d",n);

/* ascunde cursor */
 _AH=1;
 _CH=0x20;
 geninterrupt(0x10);

} /* sfirsit fereastra */

void orizontal(int a,int chenar)
/* traseaza un chenar orizontal */
{
 while(a--)
     switch(chenar){
         case SIMPLU:
             putch(196);
             break;
         case DUBLU:
             putch(205);
             break;
     }
}
```

```
void vertical(int a,int col,int lin,int chenar)
{
 while(a--) {
     gotoxy(col,lin++);
     switch(chenar){
         case SIMPLU:
             putch(179);
             break;
         case DUBLU:
             putch(186);
             break;
     }
 }
}
```

18.7 Să se scrie un program care afişează ferestre pe ecran în mod aleator. Ferestrele sînt de dimensiune fixă, dar au poziţii aleatoare pe ecran. De asemenea, ele pot avea chenar format dintr-o linie simplă sau dublă sau să nu aibă chenar. Culorile de fond şi de afişare a caracterelor sînt aleatoare. Ferestrele se numerotează şi numărul ferestrei se afişează în fereastră. Prima fereastră afişată se numerotează cu 1.

După afişarea unei ferestre se va acţiona o tastă oarecare, corespunzătoare codului ASCII, pentru a afişa fereastra următoare. Se revine la fereastra precedentă dacă se acţionează tasta ESC.

Execuţia programului se termină în cazul în care se acţionează tasta ESC în momentul în care nu este afişată nici o fereastră.

Se presupune că se dispune de un adaptor color EGA/VGA.

PROGRAMUL BXVIII7

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <alloc.h>
#include <dos.h>

#define MAX 100
#define ESC 0x1b
#define SIMPLU 1
#define DUBLU 2
```

```
typedef struct {
 int x,y,u,v;
 void far *zonfer;
} ELEM;
ELEM far *stiva[MAX];

int istiva;

#include "bviii2.cpp"  /* pcit_int     */
#include "bviii3.cpp"  /* pcit_int_lim */
#include "bxviii6.cpp" /* fereastra    */

main() /* afiseaza ferestre pe ecran in mod aleator */
{
 int c,culoare,fond,i,inalt,j,lung,s,stanga,sus;
 struct text_info info,crt;
 struct time ora_crt;

 istiva=0;
 clrscr();

/* pastreaza tot ecranul */
 if((stiva[istiva]=
              (ELEM *)farmalloc(sizeof(ELEM)))==0){
    printf("memorie insuficienta\n");
    exit(1);
 }
 if((stiva[istiva]->zonfer=farmalloc(2*80*25))==0){
    printf("memorie insuficienta\n");
    exit(1);
 }
 if(gettext(1,1,80,25,stiva[istiva]->zonfer)==0){
    printf("nu se poate salva ecranul\n");
    exit(1);
 }
 istiva++;

/* salveaza parametrii ecranului */
 gettextinfo(&info);

/* citeste dimensiunile ferestrelor: lungimea si inaltimea */
 if(pcit_int_lim("lungime:",8,70,&lung)==0){
    printf("s-a tastat EOF\n");
```

```
    exit(1);
 }
 if(pcit_int_lim("inaltime:",5,15,&inalt)==0){
    printf("s-a tastat EOF\n");
    exit(1);
 }

/* coordonatele maxime pentru coltul din stinga sus a ferestrelor */
 i=79-lung;
 j=25-inalt;

/* seteaza saminta pentru sirul de numere pseudo-aleatoare care
   definesc parametrii ferestrelor:
      - coltul din stinga sus;
      - atributul de culoare. */
 gettime(&ora_crt);
 s=(3600L*ora_crt.ti_hour+60*ora_crt.ti_min+
          ora_crt.ti_sec)%65535;
 srand(s);
 printf("Actionati o tasta pentru a continua\n");
 printf("cu ESC se revine la ecranul precedent\n");
 for(;;){
    c=getch();
    if(c!=ESC){

/* s-a tastat un caracter diferit de ESC */
/* se genereaza parametrii ferestrei */
       stanga=random(i)+1;
       sus=random(j)+1;
       fond=random(8);
       while((culoare=random(15)+1)==fond)
          ;
       fereastra(stanga,sus,stanga+lung,sus+inalt,
                 fond,culoare,istiva%3,istiva);
       continue;
    }

/* s-a tastat ESC */
    if(--istiva>0){

/* se reface zona din ecran eliminind fereastra activa */
       puttext(stiva[istiva]->x,stiva[istiva]->y,
              stiva[istiva]->u,stiva[istiva]->v,
```

```
        stiva[istiva]->zonfer);
        farfree(stiva[istiva]);
    }
    else
```

/* se intrerupe executia programului */

```
        break;
}
puttext(info.winleft,info.wintop,info.winright,
        info.winbottom,stiva[0]);
window(1,1,80,25);
farfree(stiva[0]);
textattr(info.attribute);
```

/* afiseaza cursorul */

```
_AH=1;
_CH=6;
_CL=7;
geninterrupt(0x10);
clrscr();
}
```

19. GESTIUNEA ECRANULUI ÎN MOD GRAFIC

Modul *grafic* presupune că ecranul este format din "puncte luminoase" (*pixeli*). Numărul acestora depinde de adaptorul grafic şi se numeşte *rezoluţie*. O rezoluţie este cu atît mai bună cu cît este mai mare.

Adaptorul CGA are o rezoluţie de 200 de rînduri a 640 de coloane, iar adaptorul EGA oferă o rezoluţie de tot atîtea coloane, dar de 350 de rînduri.

Adaptorul VGA oferă o rezoluţie de 480 de rînduri a 640 de coloane sau chiar mai mare, de pînă la 768 de rînduri, a 1024 de coloane.

Pentru gestiunea ecranului în mod grafic se pot utiliza peste 60 de funcţii standard aflate în biblioteca sistemului. Aceste funcţii au prototipul în fişierul *graphics.h*.

În acest capitol indicăm funcţiile mai importante care permit gestiunea ecranului în mod grafic.

Aceste funcţii folosesc pointeri de tip *far* (32 de biţi).

19.1. Setarea modului grafic

Modul grafic se setează cu ajutorul funcţiei *initgraph*. Această funcţie poate fi folosită singură sau împreună cu o altă funcţie numită *detectgraph* care determină parametrii adaptorului grafic. Prototipul ei este:

void far detectgraph(int far ****gd*, int far ****gm***);**

unde:

- În zona spre care pointează *gd* se păstrează una din valorile:

Constantă simbolică	**Valoare**
CGA	1
MCGA	2
EGA	3
EGA64	4
EGAMONO	5
IBM8514	6
HERCMONO	7
ATT400	8

VGA	9
PC3270	10

- În zona spre care pointează *gm* se memorează una din valorile:

Pentru CGA:

Constantă simbolică	**Valoare**
CGAC0	0
CGAC1	1
CGAC2	2
CGAC3	3

Toate aceste valori corespund unei rezoluţii de 320*200 puncte şi permit maximum 4 culori.

CGAHI	4

Permite o rezoluţie de 640*200 puncte şi lucrează numai în alb/negru.

Pentru EGA:

Constantă simbolică	**Valoare**
EGALO	0

Are o rezoluţie de 640*200 puncte şi permite maximum 16 culori.

EGAHI	1

Are o rezoluţie de 640*350 puncte.

Pentru VGA:

Constantă simbolică	**Valoare**
VGALO	0

Are o rezoluţie de 640*200 puncte.

VGAMED	1

Are o rezoluţie de 640*350 puncte.

VGAHI	2

Are o rezoluţie de 640*480 puncte.

Valorile spre care pointează *gd* definesc nişte funcţii standard corespunzătoare adaptorului grafic. Aceste funcţii se numesc *drivere*. Ele se află în subdirectorul BGI.

Funcţia *detectgraph* detectează adaptorul grafic prezent la calculator şi păstrează valoarea corespunzătoare acestuia în zona spre care pointează *gd*.

Modul grafic se defineşte în aşa fel încît el să fie cel mai performant pentru adaptorul grafic curent.

Cele mai utilizate adaptoare sînt cele de tip EGA şi VGA. De aceea, în cele ce urmează, vom presupune că adaptorul curent este de tip EGA. În felul acesta, toate exerciţiile din acest capitol pot fi rulate la un calculator echipat cu un adaptor EGA sau VGA.

Apelul funcţiei *detectgraph* trebuie să fie urmat de apelul funcţiei *initgraph*. Aceasta setează modul grafic în conformitate cu parametri stabiliţi de apelul prealabil al funcţiei *detectgraph*.

Funcţia *initgraph* are prototipul:

**void far initgraph(int far ****gd*, **int far ****gm*, **int far ****cale*);

unde:

gd şi gm	- Sînt pointeri care au aceeaşi semnificaţie ca în cazul funcţiei *detectgraph*.
cale	- Este pointer spre şirul de caractere care defineşte calea subdirectorului BGI care conţine driverele. De exemplu, dacă BGI este subdirector al directorului BORLANDC, atunci vom folosi şirul de caractere: "c:\\BORLANDC\\BGI"

Exemplu:

Pentru setarea în mod implicit a modului grafic, putem utiliza secvenţa de mai jos:

```
int driver, mod_grafic;
...
detectgraph(&driver,&mod_grafic);
initgraph(&driver,&mod_grafic,
          "c:\\BORLANDC\\BGI");
```

După apelul funcţiei *initgraph* se pot utiliza celelalte funcţii standard de gestiune grafică a ecranului.

Din modul grafic se poate ieşi apelînd funcţia *closegraph* de prototip:

void far closegraph(**void**);

Funcţia *initgraph* poate fi apelată folosind secvenţa de mai jos:

```
int driver, mod_grafic;
...
driver = DETECT;
initgraph(&driver,&mod_grafic,
          "c:\\BORLANDC\\BGI");
```

Constanta simbolică DETECT este definită în fişierul *graphics.h* alături de celelalte constante simbolice care definesc driverul. Aceasta are valoarea zero.

Prin apelul de mai sus, funcţia *initgraph* apelează funcţia *detectgraph* pentru a defini parametrii impliciţi ai adaptorului grafic.

Utilizatorul poate defini el însuşi parametri pentru iniţializarea modului grafic. De exemplu, secvenţa:

```
int driver = 1; /* CGA */
int mod_graf = 0; /* CGAC0 */
initgraph(&driver,&mod_graf, "c:\\BORLANDC\\BGI");
```

setează modul grafic corespunzător unui adaptor grafic CGA cu rezoluţia 320*200 puncte.

În afara acestor funcţii, utilizatorul mai poate utiliza şi funcţia *setgraphmode* care selectează un mod grafic diferit de cel activat implicit prin *initgraph*. Această funcţie are prototipul:

void far setgraphmode(**int** *mode*);

unde:

mode - Are valorile:

0 - 4	pentru CGA;
0 - 1	pentru EGA;
0 - 2	pentru VGA.

Ea poate fi utilizată împreună cu funcţia *restorecrtmode* de prototip:

void far restorecrtmode(**void**);

Această funcţie permite revenirea la modul precedent, iar *setgraphmode* realizează trecerea inversă.

Alte funcţii din această categorie sînt:

void far graphdefaults(**void**);

repune parametri grafici la valorile implicite;

int far getgraphmode(void);

returnează codul modului grafic;

char *far getmodename(int *mod*);

returnează pointerul spre numele modului grafic definit de codul numeric *mod*;

char *far getdrivername(void);

returnează pointerul spre numele driverului corespunzător adaptorului curent;

void far getmoderange(int *grafdriv*, int far **min*, int far **max*);

defineşte valorile minimale şi maximale ale modului grafic utilizat;

grafdriv are ca valoare una din valorile 1-10 indicate mai sus la funcţia *detectgraph*.

Valoarea minimă a modului grafic este păstrată în zona spre care pointează *min*, iar cea maximă în zona spre care pointează *max*.

Exerciţii:

19.1 Să se scrie un program care setează modul grafic în două feluri:

- cu ajutorul funcţiei *detectgraph*;
- fără această funcţie.

Programul afişează rezultatele setării.

PROGRAMUL BXIX1

```
#include <conio.h>
#include <stdio.h>
#include <graphics.h>

main() /*seteaza modul grafic si afiseaza parametrii setarii */
{
 int gdriv,gmod;
 int mod,min,max;

/* setare prin apelul functiei detectgraph */
 detectgraph(&gdriv,&gmod);
 printf("valori dupa apelul functiei\
       detectgraph\n");
```

```
printf("driver=%d\tmod grafic=%d\n",gdriv,gmod);
printf("Actionati o tasta pentu a continua\n");
getch();
initgraph(&gdriv,&gmod,"c:\\borlandc\\bgi");
printf("valori dupa initgraph\n");
printf("driver=%d\tmod grafic=%d\n",gdriv,gmod);
printf("Actionati o tasta pentru a continua\n");
getch();
closegraph();

/* setare fara apelul lui detectgraph */
gdriv=DETECT;
initgraph(&gdriv,&gmod,"c:\\borlandc\\bgi");
printf("initializare fara detectgraph\n");
printf("driver=%d\tmod grafic=%d\n",gdriv,gmod);
printf("Actionati o tasta pentru a continua\n");
getch();

/* afiseaza numele adaptorului grafic curent */
printf("adaptor grafic: %s\n",getdrivername());
mod=getgraphmode();
printf("cod mod=%d\tmod grafic:%s\n",mod,
      getmodename(mod));
getmoderange(gdriv,&min,&max);
printf("domeniul pentru adaptorul\
      grafic=[%d,%d]\n",min,max);
printf("Actionati o tasta pentru a continua\n");
getch();
closegraph();
}
```

19.2. Gestiunea culorilor

Adaptoarele grafice sînt prevăzute cu o zonă de memorie în care se păstrează date specifice gestiunii ecranului. Această zonă de memorie poartă denumirea de *memorie video*.

În mod grafic, ecranul se consideră format din puncte luminoase numite *pixeli*. Poziţia pe ecran a unui pixel se defineşte printr-un sistem binar:

(x,y)

unde:

x - Defineşte coloana în care este afişat pixelul.

y - Defineşte linia în care este afişat pixelul.

În cazul adaptoarelor color, unui pixel îi corespunde o culoare.

Culoarea pixelilor se păstrează pe biţi în memoria video. Memoria video necesară pentru a păstra starea ecranului setat în mod grafic, se numeşte *pagină video*. Adaptoarele pot conţine mai multe pagini video.

Gestiunea culorilor este dependentă de tipul de adaptor grafic existent la microprocesor.

În cele ce urmează vom avea în vedere adaptoarele grafice de tip EGA/VGA.

În mod concret, ne vom referi la facilităţile oferite de adaptorul EGA, deoarece adaptorul VGA, avînd performanţe superioare, permite utilizarea acestor facilităţi.

Numărul maxim al culorilor care pot fi afişate cu ajutorul unui adaptor EGA este de 64.

Culorile se codifică prin numere întregi din intervalul [0,63].

Cele 64 de culori nu pot fi afişate simultan pe ecran. În cazul adaptorului EGA se pot afişa simultan pe ecran cel mult 16 culori. Mulţimea culorilor care pot fi afişate simultan pe ecran se numeşte *paletă*. Culorile din componenţa unei palete pot fi modificate de utilizator prin intermediul funcţiilor standard. La iniţializarea modului grafic se setează o paletă *implicită*.

Paleta se defineşte cu ajutorul unui tablou de 16 elemente pentru adaptorul EGA. Elementele acestui tablou au valori din intervalul [0,63]. Fiecare element din acest tablou reprezintă codul unei culori.

Codurile culorilor din paleta implicită au denumiri simbolice definite în fişierul *graphics.h*. Ele au prefixul EGA_.

În tabela de mai jos se indică codurile culorilor pentru paleta implicită.

Funcţiile de gestiune a culorilor pot avea ca parametri nu numai codurile culorilor, ci şi indecşi în tabloul care defineşte culorile unei palete. De aceea, indicii din intervalul [0,15] pot fi referiţi prin constante simbolice definite în fişierul *graphics.h*. Aceste denumiri sugerează culoarea din compunerea paletei.

Tabela paletei implicite

indice		codul culorilor	
denumire simbolică	**valoare**	**denumire simbolică**	**valoare**
BLACK	0	EGA_BLACK	0
BLUE	1	EGA_BLUE	1
GREEN	2	EGA_GREEN	2
CYAN	3	EGA_CYAN	3
RED	4	EGA_RED	4
MAGENTA	5	EGA_MAGENTA	5
BROWN	6	EGA_BROWN	20
LIGHTGRAY	7	EGA_LIGHTGRAY	7
DARKGRAY	8	EGA_DARKGRAY	56
LIGHTBLUE	9	EGA_LIGHTBLUE	57
LIGHTGREEN	10	EGA_LIGHTGREEN	58
LIGHTCYAN	11	EGA_LIGHTCYAN	59
LIGHTRED	12	EGA_LIGHTRED	60
LIGHTMAGENTA	13	EGA_LIGHTMAGENTA	61
YELLOW	14	EGA_YELLOW	62
WHITE	15	EGA_WHITE	63

Culoarea fondului (*background*) este totdeauna cea corespunzătoare indicelui zero.

Culoarea pentru desenare (*foreground*) este cea corespunzătoare indicelui 15.

Culoarea de fond poate fi modificată cu ajutorul funcţiei *setbkcolor*. Aceasta are prototipul:

void far setbkcolor(int *culoare***);**

unde:

culoare - Este index în tabloul care defineşte paleta.

De exemplu, dacă se utilizează apelul:

setbkcolor(BLUE);

atunci culoarea de fond devine albastră.

Pentru a cunoaşte culoarea de fond curentă se poate apela funcţia *getbkcolor* de prototip:

int far getbkcolor(void);

Ea returnează indexul în tabloul care defineşte paleta pentru culoarea de fond.

Culoarea pentru desenare poate fi modificată folosind funcţia *setcolor* de prototip:

void far setcolor(int *culoare***);**

unde:

culoare - Este index în tabloul care defineşte paleta.

De exemplu, dacă se utilizează apelul:

setcolor(YELLOW);

atunci culoarea pentru desenare este galbenă.

Culoarea pentru desenare se poate determina apelînd funcţia *getcolor* de prototip:

int far getcolor(void);

Ea returnează indexul în tabloul care defineşte paleta relativ la culoarea pentru desenare.

Paleta curentă poate fi modificată folosind funcţiile *setpalette* şi *setallpalette*.

Prima se foloseşte pentru a modifica o culoare din paleta curentă. Ea are prototipul:

void far setpalette(int *index***, int** *cod***);**

unde:

index - Este un întreg din intervalul [0,15] şi reprezintă indexul în tabloul care defineşte paleta pentru culoarea care se modifică.

cod - Este un întreg din intervalul [0,63] şi reprezintă codul culorii care o înlocuieşte în paletă pe cea veche.

De exemplu, apelul:

setpalette(DARKGRAY,45);

modifică culoarea corespunzătoare indicelui DARKGRAY (adică 8), prin culoarea de cod 45.

Cealaltă funcţie permite modificarea simultană a mai multor culori din compunerea paletei. Ea are prototipul:

void far setallpalette(struct palettetype far ****paleta*);

unde:

palettetype - Este un tip definit în fişierul *graphics.h* ca mai jos:

```
struct palettetype {
        unsigned char size;
        signed char colors[MAXCOLORS +1];
};
```

unde:

size - Este dimensiunea paletei.

colors - Este un tablou ale cărui elemente au ca valori codurile culorilor componente ale paletei care se defineşte.

Modificarea paletei curente cu ajutorul funcţiei *setpalette* sau *setallpalette* conduce la schimbarea corespunzătoare a culorilor afişate pe ecran în momentul apelului funcţiilor respective.

Pentru a determina codurile culorilor componente ale paletei curente vom folosi funcţia *getpalette* de prototip:

void far getpalette(struct palettetype far ****paleta*);

Exemplu:

```
struct palettetype pal;
...
getpalette(&pal);
...
```

După apelul funcţiei *getpalette* se atribuie componentelor structurii datele corespunzătoare din paleta curentă.

Paleta implicită poate fi determinată folosind funcţia *getdefaultpalette* de prototip:

struct palettetype *far getdefaultpalette(void);

Numărul culorilor dintr-o paletă poate fi obţinut apelînd funcţia *getmaxcolor* de prototip:

int far getmaxcolor(void);

Funcţia returnează numărul maxim de culori diminuat cu 1. Deci, în cazul adaptorului EGA funcţia returnează valoarea 15.

O altă funcţie care determină dimensiunea paletei este funcţia *getpalettesize*. Ea are prototipul:

int far getpalettesize(void);

Funcţia returnează numărul culorilor componente ale paletei. În cazul adaptorului EGA funcţia returnează valoarea 16.

Exerciţii:

19.2 Să se scrie un program care afişează codurile culorilor pentru paleta implicită.

PROGRAMUL BXIX2

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>

main() /* afiseaza codurile culorilor pentru paleta implicita */
{
 int gd=DETECT,gm,i;
 struct palettetype far *pal=(void *)0;

 initgraph(&gd,&gm,"c:\\borlandc\\bgi");
 pal=getdefaultpalette();
 for(i=0;i<16;i++){
    printf("colors[%d]=%d\n",i,pal->colors[i]);
    getch();
 }
 closegraph();
}
```

19.3. Starea ecranului

În mod grafic, ecranul se compune din n*m puncte luminoase (pixeli). Aceasta înseamnă că pe ecran se pot afişa *m* linii a *n* pixeli fiecare.

Poziţia unui pixel se defineşte printr-un sistem binar de întregi:

(x,y)

numite coordonatele pixelului. Coordonata *x* defineşte coloana pixelului, iar *y* defineşte linia acestuia.

Pixelul aflat în colţul din stînga sus are coordonatele (0,0).

Coloanele se numerotează de la stînga spre dreapta, iar liniile de sus în jos.

Biblioteca grafică a sistemului conţine 4 funcţii care permit utilizatorului să obţină următoarele informaţii relativ la ecran:

- coordonata maximă pe orizontală;
- coordonata maximă pe verticală;
- poziţia curentă (pixel curent).

Prototipurile acestor funcţii sînt:

int far getmaxx(void);

funcţia returnează coordonata maximă pe orizontală (abscisa maximă);

int far getmaxy(void);

funcţia returnează coordonata maximă pe verticală (ordonata maximă);

int far getx(void);

funcţia returnează poziţia pe orizontală (abscisa) a pixelului curent;

int far gety(void);

funcţia returnează poziţia pe verticală (ordonata) a pixelului curent.

Exerciţii:

19.3 Să se scrie un program care afişează următoarele informaţii:

- culoarea fondului;
- culoarea pentru desenare;
- coordonatele maxime;
- coordonatele pixelului curent.

PROGRAMUL BXIX3

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>

main() /* afiseaza:
              - culoarea fondului;
              - culoarea pentru desenare;
              - coordonatele maxime;
              - coordonatele pixelului curent. */
{
 int gd=DETECT,gm,culf,culd,crtx,crty,maxx,maxy;
```

```
 initgraph(&gd,&gm,"c:\\borlandc\\bgi");
 culf=getbkcolor();
 culd=getcolor();
 maxx=getmaxx();
 maxy=getmaxy();
 crtx=getx();
 crty=gety();
 closegraph();
 printf("culoarea fondului=%d\n",culf);
 printf("culoarea pentru desenare=%d\n",culd);
 printf("abscisa maxima=%d\n",maxx);
 printf("ordonata maxima=%d\n",maxy);
 printf("abscisa curenta=%d\n",crtx);
 printf("ordonata curenta=%d\n",crty);
 printf("Actionati o tasta pentru a continua\n");
 getch();
 closegraph();
}
```

19.4. Gestiunea textelor

Afişarea textelor presupune definirea unor parametri care pot fi gestionaţi prin funcţiile descrise mai jos.

În mod grafic dispunem de mai multe seturi de caractere. Setul de caractere se alege prin intermediul parametrului numit *font*. Pentru acest parametru se pot utiliza următoarele valori:

Constantă simbolică	**Valoare**
DEFAULT_FONT	0
TRIPLEX_FONT	1
SMALL_FONT	2
SANS_SERIF_FONT	3
GOTHIC_FONT	4

Alţi parametri utilizaţi în definirea caracterelor sînt:

- înălţimea şi lăţimea caracterelor;
- direcţia de scriere a caracterelor:
- • de la stînga la dreapta: HORIZ_DIR;

- de jos în sus: VERT_DIR;

– cadrajul caracterelor faţă de poziţia curentă:

- pe orizontală; poziţia curentă se află:
  - » în stînga: LEFT_TEXT;
  - » în centru: CENTER_TEXT;
  - » în dreapta: RIGHT_TEXT.
- pe verticală; poziţia curentă se află în:
  - » marginea inferioară: BOTTOM_TEXT;
  - » centru: CENTER_TEXT;
  - » marginea superioară: TOP_TEXT.

Aceşti parametri se setează cu ajutorul a două funcţii *settextstyle* şi *settextjustify*. Prima are prototipul:

void far settextstyle(int *font*, **int** *direction*, **int** *charsize*);

unde:

font - Defineşte setul de caractere.

direction - Defineşte direcţia de scriere a textului.

charsize - Defineşte dimensiunea caracterului în pixeli.

Dimensiunea se defineşte astfel:

Valoarea parametrului	Matricea folosită pentru afişarea caracterului (în pixeli)
1	8*8
2	16*16
3	24*24
...	...
10	80*80

Dimensiunea poate fi stabilită de utilizator folosind funcţia *setusercharsize*. În acest caz, parametrul *charsize* are valoarea zero.

Cea dea doua funcţie defineşte cadrajul textului. Ea are prototipul:

void far settextjustify(int *oriz*, **int** *vert*);

unde:

oriz - Defineşte cadrajul pe orizontală;

vert - Defineşte cadrajul pe verticală.

Valorile acestor parametri pot fi determinate cu ajutorul funcţiei *gettextsettings* de prototip:

void far gettextsettings(struct textsettingstype far ****textinfo*);**

Tipul *textsettingstype* este definit în fişierul *graphics.h* astfel:

```
struct textsettingstype {
        int font;
        int direction;
        int charsize;
        int horiz;
        int vert;
};
```

Funcţia *gettextsettings* atribuie componentelor structurii de tip *textsettingstype* valorile curente.

Amintim valorile numerice ale constantelor simbolice indicate mai sus.

Constantă simbolică	**Valoare**
LEFT_TEXT	0
CENTER_TEXT	1
RIGHT_TEXT	2
BOTTOM_TEXT	0
TOP_TEXT	2
HORIZ_DIR	0
VERT_DIR	1

După setarea parametrilor de mai sus se pot afişa texte folosind funcţiile *outtext* şi *outtextxy*. Prima afişează textul începînd cu poziţia curentă de pe ecran. Cea de a doua funcţie permite afişarea textului începînd cu un punct al cărui coordonate sînt definite prin primii doi parametri efectivi ai funcţiei.

Funcţiile afişează caractere colorate folosind culoarea de desenare curentă.

Funcţia *outtext* are prototipul:

void far outtext(char far ****sir*);**

unde:

sir - Este pointer spre o zonă de memorie în care se păstrează caracterele de afişat.

Se afişează caracterele respective pînă la întîlnirea caracterului NUL.

Funcţia *outtextxy* are prototipul:

void far outtextxy(int *x*, **int** *y*, **char far** ***sir**);

unde:

(x,y) - Defineşte poziţia punctului de pe ecran începînd cu care se afişează textul.

sir - Are aceeaşi semnificaţie ca în cazul funcţiei *outtext*.

Dimensiunile în pixeli a unui şir de caractere se pot determina folosind funcţiile *textheight* şi *textwidth*. Acestea au prototipurile:

int far textheight(char far ***sir**);

funcţia returnează înălţimea în pixeli a şirului păstrat în zona spre care pointează *sir*;

int far textwidth(char far ***sir**);

funcţia returnează lăţimea în pixeli a şirului aflat în zona spre care pointează *sir*.

Utilizatorul poate defini dimensiunea caracterelor apelînd funcţia *setusercharsize*. Ea are prototipul:

void far setusercharsize(int *multx*, **int** *divx*, **int** *multy*, **int** *divy*);

Dimensiunea caracterelor se defineşte prin înmulţirea lăţimii lor cu multx/divx şi a înălţimii cu multy/divy.

Modificarea dimensiunii caracterelor în acest mod este posibilă pentru fonturile diferite de fontul DEFAULT_FONT.

Observaţie:

Funcţia *printf* poate fi folosită pentru a afişa caractere în mod obişnuit. Ea ignoră parametri indicaţi mai sus.

Exerciţii:

19.4 Să se scrie un program care afişează texte folosind toate cele 5 fonturi, caracterele avînd pe rînd dimensiunile 1, 2, 3 şi 4. În cazul fonturilor diferite de DEFAULT_FONT, se vor afişa texte ale căror caractere se vor afla în rapoartele:

- 4/3 în lăţime;
- 2/1 în înălţime.

PROGRAMUL BXIX4

```
#include <graphics.h>
#include <conio.h>
#include <stdio.h>
#include <string.h>

main() /* afiseaza texte cu diferite fonturi si dimensiuni */
{
 int gdriver=DETECT,gmod;
 char *denfont[]={
     "Val=0 DEFAULT_FONT",
     "Val=1 TRIPLEX_FONT",
     "Val=2 SMALL_FONT",
     "Val=3 SANS_SERIF_FONT",
     "Val=4 GOTHIC_FONT"
 };
 int stil,x=0,y=0;
 int dim;
 char dimensiune[30];

 for(dim=1;dim<5;dim++){
     y=0;
     initgraph(&gdriver,&gmod,"c:\\borlandc\\bgi");
     for(stil=DEFAULT_FONT;stil<=GOTHIC_FONT;stil++)
     {
         settextstyle(stil,HORIZ_DIR,dim);
         sprintf(dimensiune,"dim=%d font:",dim);
         outtextxy(x,y,dimensiune);
         x += textwidth(dimensiune); /* avans la coloana
                                  libera de pe aceeasi linie */
         outtextxy(x,y,denfont[stil]);
         y += textheight(denfont[stil]); /* avans la
                                  inceputul liniei urmatoare */
         x=0;
/* se defineste dimensiunea de catre utilizator
     raport: 4/3 in latime;
             2/1 in lungime. */
         if(stil!=DEFAULT_FONT){
             setusercharsize(4,3,2,1); /* defineste
                 raporturile pentru dimensiunile caracterelor */
             strcpy(dimensiune,
                 "dim utilizator:4/3,2/1");
```

```
            outtextxy(x,y,dimensiune);
            y+=textheight(dimensiune);
            x=0;
        }
        getch();
    }
    closegraph();
    printf("Actionati o tasta pentru a\
        continua\n");
    getch();
}
}
```

19.5 Să se scrie un program care afişează texte cadrate în toate variantele definite de funcţia *settextjustify*.

PROGRAMUL BXIX5

```
#include <graphics.h>
#include <conio.h>
#include <stdio.h>

main() /* afiseaza texte cadrate in toate variantele definite de functia
        settextjustify */
{
 int gdriver=DETECT,gmod;
 char *hjust[]={"LEFT_TEXT",
     "CENTER_TEXT",
     "RIGHT_TEXT",
 };
 char *vjust[]={"BOTTOM_TEXT",
     "CENTER_TEXT",
     "TOP_TEXT"
 };
 int x=200,y=100,hj,vj;

 for(hj=LEFT_TEXT;hj<=RIGHT_TEXT;hj++) {
     initgraph(&gdriver,&gmod,"c:\\borlandc\\bgi");
     outtextxy(x,y,hjust[hj]);
     y+=textheight(hjust[hj])+8;
     for(vj=BOTTOM_TEXT;vj<=TOP_TEXT;vj++){
         settextjustify(hj,vj);
         outtextxy(x,y,"N");
```

```
            x+=textwidth("N");
            outtextxy(x+100*vj+100,y,vjust[vj]);
            getch();
        }
        closegraph();
    }
}
```

19.5. Gestiunea imaginilor

În paragrafele precedente s-a arătat că ecranul în mod grafic se compune din n*m puncte luminoase numite *pixeli*. Un pixel are o poziţie definită prin coordonatele sale şi este colorat în cazul adaptoarelor color.

La adaptoarele monocrom pixelul are o nuanţă de gri.

Utilizatorul poate afişa pe ecran un pixel cu ajutorul funcţiei *putpixel*. Aceasta are prototipul:

void far putpixel(int *x*, **int** *y*, **int** *culoare*);

unde:

(x,y)	- Defineşte poziţia punctului.
culoare	- Defineşte culoarea punctului.
	- Este un întreg din intervalul [0,15] şi reprezintă indicele culorii în tabela care defineşte paleta curentă.

Funcţia *getpixel* permite stabilirea culorii unui pixel afişat pe ecran. Ea are prototipul:

unsigned far getpixel(int *x*, **int** *y*);

unde:

(x,y) - Defineşte poziţia punctului.

Funcţia returnează un întreg din intervalul [0,15]. Acesta defineşte culoarea pixelului de coordonate (x,y), fiind index în tabloul care defineşte paleta curentă.

Ecranul poate fi partajat în mai multe părţi care pot fi gestionate independent. Aceste părţi le vom numi *ferestre grafice*.

În continuare, prin ferestră vom înţelege o fereastră grafică. O fereastră se defineşte cu ajutorul funcţiei *setviewport* de prototip:

void far setviewport(int *st*, **int** *sus*, **int** *dr*, **int** *jos*, **int** *d*);

unde:

(st,sus) - Sînt coordonatele colţului stînga sus al ferestrei.

(dr,jos) - Sînt coordonatele colţului dreapta jos al ferestrei.

d - Indicator cu privire la decuparea desenului (vezi mai jos).

Fereastra definită în urma apelului funcţiei *setviewport* devine fereastra activă. Iniţial (imediat după setarea modului grafic), fereastra activă este tot ecranul.

O fereastră activă se poate şterge cu ajutorul funcţiei *clearviewport*. Ea are prototipul:

void far clearviewport(**void**);

După apelul funcţiei *clearviewport*, toţi pixelii ferestrei active au aceeaşi culoare şi anume culoarea de fond curentă.

Poziţia curentă după apelul funcţiei *clearviewport* este pixelul de coordonate relative (0,0), adică chiar colţul din stînga sus al ferestrei.

O altă funcţie utilizată pentru a şterge tot ecranul este funcţia *cleardevice*. Ea are prototipul:

void far cleardevice(**void**);

După apelul funcţiei *cleardevice* se şterge tot ecranul şi pixelul curent devine cel din colţul stînga sus al ecranului.

Parametri ferestrei active se pot determina apelînd funcţia *getviewsettings*. Aceasta are prototipul:

void far getviewsettings(**struct viewporttype far** **fereastra*);

unde:

viewporttype - Este un tip definit în fişierul *graphics.h* astfel:

```
struct viewporttype {
        int left;
        int top;
        int right;
        int bottom;
        int clip;
};
```

După apelul funcţiei *getviewsettings*, la componentele structurii de tip *viewporttype* de la apel li se atribuie valorile corespunzătoare ale ferestrei active.

Parametrul *clip* are două valori:

CLIP_ON (valoarea 1)

şi

CLIP_OFF (valoarea zero).

Dacă *clip* are valoarea 1, atunci funcţiile de afişare a textelor şi de desenare nu pot scrie sau desena în afara limitelor ferestrei active. În caz contrar, se pot depăşi limitele ferestrei active. Deci textele şi figurile care nu încap în fereastra activă se trunchiază dacă:

clip = CLIP_ON.

Imaginea ecranului se păstrează în memoria video a adaptorului grafic şi formează o pagină. În cazul adaptoarelor de tip EGA/VGA, adaptorul dispune de o memorie video capabilă să memoreze 8 pagini. Acestea se numerotează de la 0 la 7.

Funcţiile de desenare şi scriere de texte acţionează asupra unei singure pagini. Aceasta se numeşte pagina activă. Utilizatorul poate activa o pagină folosind funcţia *setactivepage* de prototip:

void far setactivepage(int *nrpag*);

unde:

nrpag - Este numărul paginii care se activează.

De obicei, pagina activă este vizualizată pe ecran. Cu toate acestea, programatorul are posibilitatea să vizualizeze o altă pagină decît cea activă. Aceasta se realizează utilizînd funcţia *setvisualpage* de prototip:

void far setvisualpage(int *nrpag*);

unde:

nrpag - Este numărul paginii care se vizualizează.

Această funcţie poate fi utilă pentru animaţie.

Imaginea unei zone dreptunghiulare de pe ecran poate fi salvată în memorie folosind funcţia *getimage*. Ea are prototipul:

void far getimage(int *st*, **int** *sus*, **int** *dr*, **int** *jos*, **void far** **zt*);

unde:

(st,sus) - Defineşte coordonatele colţului stînga sus a zonei de pe ecran care se salvează.

(dr,jos) - Defineşte coordonatele colţului dreapta jos a zonei de pe ecran care se salvează.

zt	- Pointer spre zona de memorie în care se salvează imaginea de pe ecran.

Dimensiunea zonei de memorie spre care pointează *zt* trebuie să fie suficient de mare pentru a putea salva datele care definesc imaginea de pe ecran, care se salvează. Această dimensiune se poate determina folosind funcţia *imagesize* de prototip:

unsigned far imagesize(int *st*, **int** *sus*, **int** *dr*, **int** *jos*);

unde:

(st,sus)	- Defineşte coordonatele colţului din stînga sus a zonei dreptunghiulare de pe ecran.
(dr,jos)	- Defineşte coordonatele colţului din dreapta jos a zonei dreptunghiulare de pe ecran.

Funcţia *imagesize* se apelează înainte de a apela funcţia *getimage* pentru a stabili dimensiunea zonei de memorie necesară pentru a salva o imagine dreptunghiulară de pe ecran. Zona de memorie respectivă se poate rezerva în memoria heap ţinînd seama de valoarea returnată de funcţia *imagesize*.

Imaginea de pe ecran salvată cu ajutorul funcţiei *getimage*, poate fi afişată pe ecran în orice parte a acestuia, cu ajutorul funcţiei *putimage*. Cu această ocazie se pot face anumite operaţii asupra datelor care definesc imaginea. Funcţia are prototipul:

void far putimage(int *st*, **int** *sus*, **void far** **zt*, **int** *op*);

unde:

(st,sus)	- Defineşte coordonatele colţului stînga sus a zonei de pe ecran în care se afişează imaginea.
zt	- Pointer spre zona în care se păstrează datele care formează imaginea de afişat.
	- Aceste date au fost păstrate în zona respectivă prin intermediul funcţiei *getimage*.
op	- Defineşte operaţia între datele aflate în zona spre care pointează *zt* şi cele existente pe ecran în zona dreptunghiulară definită de parametri *st*, *sus* (colţul din stînga sus; colţul din dreapta jos rezultă din datele existente în zona spre care pointează *zt*).

Parametrul *op* se defineşte ca mai jos:

Constantă simbolică	Valoare	Acţiune
COPY_PUT	0	copiază imaginea din memorie pe ecran
XOR_PUT	1	"sau exclusiv" între datele de pe ecran şi cele aflate în memorie
OR_PUT	2	"sau" între datele de pe ecran şi cele din memorie
AND_PUT	3	"şi" între datele de pe ecran şi cele din memorie
NOT_PUT	4	copiază imaginea din memorie pe ecran complementînd datele aflate în memorie.

Pentru a defini pixelul curent se utilizează funcţia *moveto*. Aceasta are prototipul:

void far moveto(int *x*, **int** *y*);

După apelul funcţiei *moveto*, pixelul curent devine cel de coordonate (x,y).

O funcţie înrudită cu aceasta este funcţia *moverel*. Aceasta are prototipul:

void far moverel(int *dx*, **int** *dy*);

Dacă notăm cu (x,y) coordonatele pixelului curent, atunci după apelul funcţiei *moverel*, pixelul curent are coordonatele:

(x+dx, y+dy)

Observaţie:

Majoritatea funcţiilor care au ca parametri coordonate de pixel, interpretează aceste coordonate ca fiind relative faţă de fereastra activă al cărei colţ stînga sus are coordonatele(0,0). Aşa de exemplu, funcţiile:

outtextxy, putpixel, getpixel, moveto

au ca parametri coordonate relative la pixelul din colţul stînga sus al ferestrei active.

Funcţiile *getx* şi *gety* returnează abscisa, respectiv ordonata, relative la pixelul din colţul stînga sus al ferestrei active.

În schimb funcţiile *getmaxx* şi *getmaxy* returnează totdeauna abscisa, respectiv ordonata, maximă pentru tot ecranul în conformitate cu modul

grafic.

De asemenea, funcţiile *setviewport* şi *getviewsettings* gestionează coordonate care sînt relative faţă de colţul stînga sus al ecranului şi nu faţă de fereastra activă. Astfel de coordonate le vom numi în continuare *absolute*.

Exerciţii:

19.6 Să se scrie un program care realizează următoarele:

a. Afişează, într-o zonă de dimensiune 50*50, pixeli coloraţi folosind toate culorile din paletă.

Zona se află în colţul din stînga sus al ecranului.

b. Defineşte fereastra de coordonate:

(60,0) - colţul stînga sus;

(120,60) - colţul dreapta jos.

Afişează în fereastra activă pixeli coloraţi folosind toate culorile din paletă.

c. Defineşte fereastra de coordonate:

(20,100) - colţul stînga sus;

(100,180) - colţul dreapta jos.

Afişează în fereastra activă pixeli coloraţi folosind toate culorile din paletă.

Afişează în fereastra activă textul "abcdefg" începînd cu poziţia de coordonate(1,1).

Se utilizează culoarea de index 6.

d. Se şterge fereastra activă.

Se afişează textul "abcdefg" începînd cu poziţia curentă din fereastră şi folosind culoarea de index 14.

e. Se şterge tot ecranul.

f. Se revine din modul grafic.

După realizarea fiecărui punct de mai sus se vizualizează ecranul apelînd funcţia *getch*. Pentru a continua, se acţionează o tastă oarecare.

PROGRAMUL BXIX6

```
#include <graphics.h>
```

```
#include <conio.h>

main() /* - afiseaza in trei zone ale ecranului pixeli colorati folosind
            toate culorile paletei;
          - in una din zone se afiseaza si textul "abcdefg" ;
          - in final se sterg zonele afisate. */
{
 int gd=DETECT,gm;
 int i,j,c;

 initgraph(&gd,&gm,"c:\\borlandc\\bgi");

/* zona din coltul stinga sus al ecranului */
 for(i=0;i<50;i++){
     c=i;
     for(j=0;j<50;j++,c++) {
         c=c%16;
         putpixel(i,j,c);
     }
 }
 getch();

/* defineste o fereastra si afiseaza in ea pixeli colorati */
 setviewport(60,0,120,60,1);
 for(i=0;i<50;i++){
     c=0;
     for(j=0;j<50;j++,c++){
         c=c%16;
         putpixel(i,j,c);
     }
 }
 getch();

/* defineste ultima zona si afiseaza in ea pixeli colorati */
 setviewport(20,100,100,180,1);
 for(i=0;i<40;i++){
     c=10;
     for(j=0;j<250;j++,c++){
         c=c%16;
         putpixel(i,j,c);
     }
 }
/* afiseaza textul "abcdefg" */
```

```
setcolor(6);
outtextxy(1,1,"abcdefg");
getch();

/* sterge ultima zona si apoi afiseaza acelasi text folosind culoarea de
   index 14 */
clearviewport();
setcolor(14);
outtext("abcdefg");
getch();

/* sterge tot ecranul */
cleardevice();
getch();

/* iesire din modul grafic */
closegraph();
getch();
}
```

19.7 Să se scrie un program care realizează următoarele:

a. Afişează un pixel în punctul de coordonate absolute (20,30), apoi textul:

(x,y)

unde:

x şi y - Sînt coordonatele returnate de funcţiile *getx* şi respectiv *gety*.

b. Se deplasează poziţia curentă cu valoarea relativă 30, atît pe abscisă, cît şi pe ordonată, apoi se afişează pixelul curent şi textul:

(x,y)

unde:

x şi y - Sînt valorile returnate de funcţiile *getx* şi respectiv *gety*.

c. Se defineşte fereastra de coordonate:

(200,0,420,70);

apoi se repetă secvenţele a-b de mai sus.

d. Se şterge fereastra activă.

e. Se şterge ecranul şi se iese din modul grafic.

Culoarea pentru afişare este culoarea de index maxim.

PROGRAMUL BXIX7

```
#include <graphics.h>
#include <conio.h>
#include <stdio.h>

main() /* - afiseaza:
         a- un pixel in punctul de coordonate absolute (20,30),
            apoi textul
                (x,y)
            unde x si y sînt coordonatele returnate de getx si res-
            pectiv gety;
         b- se deplaseaza pozitia curenta cu valoarea relativa
            30 atit pe abscisa cit si pe ordonata, apoi se afiseaza
            pixelul curent si textul
                (x,y)
            unde x si y se definesc ca mai sus;
         c- se defineste fereastra
                200,0,420,70
            si se repeta secventele a-b de mai sus;
         se sterg afisarile si apoi se iese din modul grafic . */

{
 int gd=DETECT,gm;
 int x,y,i;
 char msg[80];

 initgraph(&gd,&gm,"c:\\borlandc\\bgi");
 for(i=0;i<2;i++){
     moveto(20,30);
     putpixel(getx(),gety(),getmaxcolor());
     sprintf(msg,"(%d %d)",getx(),gety());
     outtext(msg);

/* deplasare relativa fata de pozitia curenta */
     moverel(30,30);
     putpixel(getx(),gety(),getmaxcolor());
     sprintf(msg,"(%d %d)",getx(),gety());
     outtext(msg);
     getch();
     setviewport(200,0,420,70,1);
 }
```

```
 clearviewport();
 getch();
 cleardevice();
 getch();
 closegraph();
}
```

19.8 Să se scrie un program care realizează următoarele:

a. Afişează pixeli coloraţi într-o zonă dreptunghiulară în colţul din stînga sus al ecranului, apoi o afişează pe ecran în zone ce au poziţii definite aleator.

b. La afişarea imaginii se folosesc toate operaţiile dintre imagini oferite de funcţia *putimage*.

c. Culoarea de fond se schimbă folosind toate culorile din paletă.

După afişările indicate mai sus execuţia se poate termina acţionînd tasta zero; orice altă tastă acţionată permite continuarea programului cu un nou set de imagini afişate aleator.

Se observă efectul operaţiilor dintre imagini cînd acestea se suprapun.

PROGRAMUL BXIX8

```
#include <graphics.h>
#include <conio.h>
#include <alloc.h>
#include <stdlib.h>
#include <dos.h>
#include <stdio.h>

main() /* - afiseaza pixeli colorati intr-o zona dreptunghiulara in
            coltul din stinga sus al ecranului, apoi afiseaza imaginea
            respectiva pe ecran in zone ce au pozitii definite aleator;
          - la afisarea imaginii se folosesc toate operatiile oferite de
            functia putimage;
          - de asemenea, se schimba culoarea de fond folosind toate
            culorile din paleta. */
{
 int gd=DETECT,gm;
 int i,j,c,k;
 unsigned dim;
 void far *buf;
 struct time t;
```

```
 int rx,ry,x,y;

 initgraph(&gd,&gm,"c:\\borlandc\\bgi");
/* afiseaza pixeli colorati */
 for(i=0;i<50;i++){
     c=i;
     for(j=0;j<50;j++,c++) {
         c=c%16;
         putpixel(i,j,c);
     }
 }
 getch();

/* salveaza zona afisata pe ecran */
 dim=imagesize(0,0,50,50);
 if((buf=farmalloc(dim))==0){
     closegraph();
     printf("Memorie insuficienta\n");
     exit(1);
 }
 getimage(0,0,50,50,buf);

/* seteaza saminta */
 gettime(&t);
 srand(t.ti_hour*3600L +t.ti_min*60+t.ti_sec);

/* se determina valorile maxime pentru abscisa si ordonata coltului sting
   al imaginilor */
 x=getmaxx()-50;
 y=getmaxy()-50;

/* afisarea aleatoare a imaginilor */
 do{
     for(k=0;k<16;k++) {
         setbkcolor(k);
         for(i=COPY_PUT;i<=NOT_PUT;i++){
             rx=random(x);
             ry=random(y);
             setviewport(0,0,getmaxx(),getmaxy(),1);
             putimage(rx,ry,buf,i);
             getch();
         }
     }
```

```
        setcolor(1); /* culoarea pentru afisarea textului in partea
                               de jos a ecranului */
        setviewport(1,340,639,349,1); /* fereastra pentru
                                                    text */
        outtextxy(0,0,"Pentru a termina tastati\
                zero; se continua cu orice alta\
                tasta");
        if(getch()=='0')
            break;
        clearviewport(); /* sterge textul afisat */
    }while(1);
    closegraph();
}
```

19.6. Tratarea erorilor

Erorile survenite în gestiunea în mod grafic a ecranului pot fi puse în evidenţă cu ajutorul funcţiei *graphresult*. Ea are prototipul:

int far graphresult(**void**);

Funcţia returnează codul ultimei erori care a apărut înaintea apelului funcţiei *graphresult* sau valoarea constantei simbolice *grOk* dacă nu au fost erori.

Constanta *grOk* este definită în fişierul *graphics.h*.

Mesajul de eroare poate fi decodificat cu ajutorul funcţiei *grapherrormsg*. Aceasta are prototipul:

char far *far grapherrormsg(**int** *coderoare*);

unde:

coderoare - Este valoarea returnată de funcţia *graphresult*.

Funcţia *grapherrormsg* returnează un pointer spre textul de eroare.

Constanta *grOk* are valoarea zero. Codurile de eroare au valori negative.

Exerciţii:

19.9 Să se scrie un program care afişează parametri impliciţi ai modului grafic setat prin apelul funcţiei initgraph:

- numele adaptorului grafic;
- numele modului grafic;
- rezoluţia;

- fereastra curentă;
- decupajul;
- poziţia curentă;
- numărul culorilor disponibile;
- culoarea curentă;
- setul de caractere;
- direcţia textului;
- dimensiunea caracterelor;
- cadrajul textelor.

PROGRAMUL BXIX9

```
#include <conio.h>
#include <graphics.h>
#include <stdio.h>
#include <stdlib.h>

int grafdriver, grafmod;
struct palettetype paleta;
int maxculori;
int xmax, ymax;
int posx,posy;
char *Fonts[]={"DefaultFont","TriplexFont",
               "SmallFont","SansSerifFont",
                "GothicFont"};
char *TextDirect[]={"HorizDir","VertDir"};
char *HorizJust[]={"LeftText","CenterText",
                   "RightText"};
char *VertJust[]={"BottomText","CenterText",
                  "TopText"};

void inimodgrafic() /* initializeaza modul grafic */
{
 int eroare;

 grafdriver=DETECT;
 initgraph( &grafdriver, &grafmod,
         "c:\\borlandc\\bgi");
 eroare=graphresult();
 if(eroare!= grOk){
    printf ("eroare la initializarea modului\
```

```
                grafic: %s\n",
        grapherrormsg(eroare));
        exit(1);
   }
   getpalette(&paleta);
   maxculori=getmaxcolor()+1;
   xmax=getmaxx();
   ymax=getmaxy();
   posx=getx();
   posy=gety();
  }

void fereastraprincipala(char *antet)
/* afiseaza antetul in fereastra principala */
{
 int inalt;

/* sterge ecranul */
 cleardevice();

/* seteaza fereastra principala pe tot ecranul */
 setviewport (0,0,xmax,ymax,1);

/* determina inaltimea textului */
 inalt=textheight("H");

/* afisarea textului */
 outtextxy(xmax/2, 2, antet );

/* seteaza o fereastra pentru a continua afisarea */
 setviewport(1, inalt+5, xmax-1,
             ymax-(inalt+ 5),1);
}

void afisinit()
/* afiseaza datele de la initializarea modului grafic */
{
 struct viewporttype infview;
 struct textsettingstype inftext;
 char *driver,*mod;
 int x,y;
 char sir[130];
```

```
getviewsettings(&infview);

gettextsettings(&inftext);
driver=getdrivername();
mod=getmodename(grafmod);

x=10;
y=4;

/* se scrie textul:rezultatele initializarii */
fereastraprincipala("Rezultatele initializarii");

/* cadrajul textului */
settextjustify(LEFT_TEXT,TOP_TEXT);

/* afiseaza adaptorul grafic */
sprintf(sir,"adaptorul grafic: %-20s(%d)",
        driver, grafdriver);
outtextxy(x,y,sir);
y+=8;

/* afiseaza modul grafic */
sprintf(sir, "modul grafic: %-20s (%d)", mod,
        grafmod);
outtextxy(x,y,sir);
y+=8;

/* afiseaza rezolutia */
sprintf(sir,"rezolutia: (0,0,%d,%d)", xmax, ymax);
outtextxy( x,y,sir);
y+=8;

/* fereastra grafica curenta */
sprintf(sir,"fereastra curenta:(%d, %d, %d, %d)",
       infview.left, infview.top,
       infview.right, infview.bottom);
outtextxy(x,y,sir);
y+=8;

sprintf(sir,"decupaj: %s",
        infview.clip ? "ON" : "OFF");
outtextxy(x,y, sir );
y+=8;
```

```
/* pozitia curenta */
sprintf(sir, "pozitia curenta:( %d, %d )",posx,
    posy);
outtextxy( x, y, sir );
y+=8;

/* numar culori disponibile */
sprintf(sir, "numar culori: %d", maxculori);
outtextxy(x, y, sir );
y+=8;

/* culoarea curenta */
sprintf(sir, "culoarea curenta: %d",getcolor());
outtextxy(x, y, sir );
y+=8;

/* setul de caractere */
sprintf(sir, "setul de caractere: %s",
        Fonts[inftext.font]);
outtextxy(x,y,sir);
y+=8;

/* directia textului */
sprintf (sir,"directia textului: %s",
         TextDirect[inftext.direction]);
outtextxy(x,y,sir);
y+=8;

/* dimensiunea caracterului */
sprintf (sir,"dimensiunea caracterului: %d",
         inftext.charsize);
outtextxy(x,y,sir);
y+=8;

/* cadraj orizontal */
sprintf(sir, "cadraj orizontal: %s",
      HorizJust[inftext.horiz]);
outtextxy(x,y,sir);
y+=8;

/* cadraj vertical */
sprintf(sir,"cadraj vertical: %s",
        VertJust[inftext.vert]);
```

```
 outtextxy(x,y,sir);
}

main() /* afiseaza starea curenta a unor parametri ai modului grafic */
{
 inimodgrafic ();
 afisinit();
 outtextxy(8,ymax-50,"Actionati o tasta pentru a\
          termina");
 getch ();
 closegraph();
}
```

19.7. Desenare şi colorare

Biblioteca standard a sistemului pune la dispoziţia utilizatorului o serie de funcţii care permit desenarea şi colorarea unor figuri geometrice.

Amintim că un punct colorat (pixel) se afişează cu ajutorul funcţiei *putpixel* de prototip (vezi paragraful 19.5):

void far putpixel(int *x*, **int** *y*, **int** *culoare*);

unde:

(x,y) - Sînt coordonatele punctului care se afişează şi ele sînt relative la fereastra activă.

culoare - Este index în tabloul care defineşte paleta curentă.

Indexul respectiv defineşte codul culorii pentru afişarea pixelului pe ecran.

Menţionăm că în acest paragraf prin parametrul *culoare* vom înţelege un index în tabloul care defineşte paleta curentă.

Acest index defineşte codul culorii pentru desenarea şi colorarea figurilor.

Pentru trasarea liniilor se pot folosi trei funcţii: *line, lineto* şi *linerel*.

Funcţia *line* are prototipul:

void far line(int *xstart*, **int** *ystart*, **int** *xfin*, **int** *yfin*);

Funcţia trasează un segment de dreaptă ale cărui capete sînt punctele de coordonate:

(xstart,ystart)

şi

(xfin,yfin).

Funcţia *lineto* are prototipul:

void far lineto(int *x*, **int** *y*);

Ea trasează un segment de dreaptă care are ca origine poziţia curentă, iar ca şi punct final cel de coordonate (x,y).

Punctul final devine poziţia curentă.

Amintim că funcţia *moveto* permite definirea poziţiei curente (vezi paragraful 19.5).

Cea de a treia funcţie utilizată la trasarea dreptelor are prototipul:

void far linerel(int *x*, **int** *y*);

Dacă notăm cu *xcrt* şi *ycrt* coordonatele poziţiei curente, atunci funcţia *linerel* trasează un segment de dreaptă ale cărui capete sînt punctele de coordonate:

(xcrt,ycrt)

şi

(xcrt+x,ycrt+y).

Alte funcţii care permit trasări de figuri geometrice utilizate frecvent sînt:

void far arc(int *xcentru*, **int** *ycentru*, **int** *unghistart*, **int** *unghifin*, **int** *raza*);

trasează un arc de cerc; unghiurile sînt exprimate în grade sexagesimale.

void far circle(int *xcentru*, **int** *ycentru*, **int** *raza*);

trasează un cerc; (xcentru,ycentru) sînt coordonatele centrului arcului de cerc şi respectiv cercului trasat de aceste funcţii; parametrul *raza* defineşte mărimea razei curbelor respective.

void far ellipse(int *xcentru*, **int** *ycentru*,
int *unghistart*, **int** *unghifin*,
int *semiaxamare*, **int** *semiaxamica*);

trasează un arc de elipsă cu centrul în punctul de coordonate (xcentru, ycentru) avînd semiaxa mare definită de parametrul *semiaxamare*, iar semiaxa mică de parametrul *semiaxamica*.

void far rectangle(int *st*, **int** *sus*, **int** *dr*, **int** *jos*);

trasează un dreptunghi definit de colţurile sale opuse:

(st,sus) - Colţul din stînga sus.

(dr,jos) - Colţul din dreapta jos.

void far drawpoly(int *nr*, **int far** **tabpct*);

trasează o linie poligonală:

nr - Numărul laturilor.

tabpct - Este un pointer spre întregi care definesc coordonatele vîrfurilor liniei poligonale.
- Acestea sînt păstrate sub forma: *abscisa_i, ordonata_i* unde *i* are valorile 1, 2, ..., nr+1.

Linia poligonală este închisă dacă primul punct coincide cu ultimul (au aceleaşi coordonate).

Coordonatele utilizate ca parametri la apelul acestor funcţii sînt relative la fereastra activă.

Culoarea de trasare este cea curentă.

La trasarea liniilor cu ajutorul funcţiilor:

line, *lineto* **şi** *linerel*

se poate defini un stil de trasare folosind funcţia *setlinestyle*. Această funcţie are prototipul:

void far setlinestyle(int *stil*, **unsigned** *sablon*, **int** *grosime*);

unde:

stil - Este întreg din intervalul [0,4] care defineşte stilul liniei conform tabelei de mai jos:

Constantă simbolică	Valoare	Stil
SOLID_LINE	0	linie continuă
DOTTED_LINE	1	linie punctată
CENTER_LINE	2	linie întreruptă formată din liniuţe de două dimensiuni
DASHED_LINE	3	linie întreruptă formată din liniuţe de aceeaşi dimensiune
USERBIT_LINE	4	stil definit de utilizator prin şablon

sablon - Defineşte stilul liniei.
- Are sens numai cînd primul parametru (stil) are valoarea

4.

- În rest este neglijat şi de aceea poate avea valoarea zero.

grosime - Defineşte lăţimea liniei în pixeli; pot fi două lăţimi:

NORM_WIDTH (valoarea 1 pixel)

şi

THICK_WIDTH (valoarea 3 pixeli).

Stilul curent poate fi determinat apelînd funcţia *getlinesettings* de prototip:

void far getlinesettings(struct linesettingstype far ***linieinfo*);

unde:

linesettingstype - Este definit în fişierul *graphics.h* astfel:

```
struct linesettingstype {
        int linestyle;
        unsigned upattern;
        int thickness;
};
```

După apelul funcţiei *getlinesettings*, componentelor structurii de tip *linesettingstype* de la apel li se atribuie valorile curente ale stilului de trasare după cum urmează:

linestyle - Are ca valoare stilul definit mai sus.

upattern - Are ca valoare şablonul definit de utilizator. Această valoare are semnificaţie numai dacă *linestyle* are valoarea 4.

thickness - Are valoarea 1 sau 3 şi reprezintă grosimea de trasare a dreptelor prin funcţiile *line, lineto* sau *linerel*.

Din cele de mai sus rezultă că pentru trasarea liniilor se pot folosi 4 stiluri *standard* (definite de valorile SOLID_LINE, DOTTED_LINE, CENTER_LINE şi DASHED_LINE), precum şi unul *nestandard*, definit de utilizator (valoarea USERBIT_LINE).

Stilul nestandard se precizează cu ajutorul parametrului *sablon*. Acesta este o dată de tip *int* (16 biţi). Fiecare bit din şablon reprezintă un pixel al liniei. Fiecare bit din şablon setat reprezintă un pixel colorat cu culoarea curentă, biţii din şablon de valoarea zero reprezintă pixeli coloraţi cu culoarea de fond.

Alte funcţii din biblioteca standard a sistemului permit trasarea de figuri

geometrice împreună cu colorarea interiorului acestora.

Indicăm mai jos prototipurile funcţiilor mai importante din această clasă:

void far bar(int *st*, **int** *sus*, **int** *dr*, **int** *jos*);

unde:

st, sus, dr şi jos - Au aceleaşi semnificaţii ca în cazul funcţiei *rectangle*.

Funcţia desenează un domeniu dreptunghiular colorat (fără a avea o frontieră scoasă în evidenţă).

void far bar3d(int *st*, **int** *sus*, **int** *dr*, **int** *jos*, **int** *profunzime*, **int** *ind*);

Funcţia desenează o prismă dreptunghiulară colorată pentru *ind* diferit de zero. Pentru ind=0, nu se trasează partea de sus a prismei.

void far pieslice(int *xcentru*, **int** *ycentru*, **int** *unghistart*,
int *unghifin*, **int** *raza*);

Funcţia desenează un sector de cerc colorat.

void far fillpoly(int *nr*, **int far** **tabpct*);

unde:

nr şi tabpct - Au aceleaşi semnificaţii ca în cazul funcţiei *drawpoly*.

Funcţia desenează un poligon colorat.

void far fillellipse(int *xcentru*, **int** *ycentru*,
int *semiaxamare*, **int** *semiaxamica*);

Funcţia desenează o elipsă colorată.

Menţionăm că figurile desenate cu funcţiile indicate mai sus se colorează folosind o culoare definită în prealabil cu ajutorul funcţiei *setfillstyle*. De asemenea, o figură poate fi colorată în mai multe feluri:

- folosind culoarea de fond;
- colorare uniformă (toţi pixelii din interiorul figurii au aceeaşi culoare) folosind culoarea definită prin funcţia *setfillstyle*;
- colorare prin haşură.

Haşura poate fi standard sau definită de utilizator.

Modul de colorare al unei figuri se defineşte tot cu ajutorul funcţiei *setfillstyle*. Ea are prototipul:

void far setfillstyle(int *hasura*, **int** *culoarea*);

unde:

hasura - Defineşte modul de colorare conform tabelului de mai jos.

culoarea - Defineşte culoarea pentru colorarea figurilor.

Parametrul *hasura* are următoarele valori:

Constantă simbolică	**Valoare**
EMPTY_FILL	0
SOLID_FILL	1
LINE_FILL	2
LTSLASH_FILL	3
SLASH_FILL	4
BKSLASH_FILL	5
LTBKSLASH_FILL	6
HATCH_FILL	7
XHATCH_FILL	8
INTERLEAVE_FILL	9
WIDE_DOT_FILL	10
CLOSE_DOT_FILL	11
USER_FILL	12

Valoarea EMPTY_FILL colorează figura folosind culoarea de fond.

Valoarea SOLID_FILL realizează colorarea uniformă a figurii (toţi pixelii din interiorul figurii au aceeaşi culoare).

Valorile de la LINE_FILL(2), pînă la CLOSE_DOT_FILL(11) definesc diferite haşuri standard.

Valoarea USER_FILL se utilizează cînd haşura se defineşte de către utilizator. Pentru a defini o haşură nestandard se utilizează funcţia *setfillpattern*. Aceasta are prototipul:

void far setfillpattern(char far **h_utilizator*, **int** *culoare*);

unde:

h_utilizator - Este un pointer spre o zonă de memorie în care se defineşte, pe 8 octeţi, haşura.

culoare - Defineşte culoarea de haşurare.

Şablonul se defineşte printr-un şir de 8 octeţi. Fiecare octet defineşte 8 pixeli, deci un şablon defineşte 64 pixeli.

Modul curent utilizat la colorare se poate determina cu ajutorul funcţiei

getfillsettings. Aceasta are prototipul:

void far getfillsettings(struct fillsettingstype far ***stilcolorare**);

unde:

fillsettingstype - Este definit în fişierul *graphics.h* astfel:

```
struct fillsettingstype {
        int pattern;
        int color;
};
```

unde:

pattern - Este componenta care defineşte modul de colorare utilizat la colorare (întreg din intervalul [0,12]).

color - Defineşte culoarea utilizată la colorare.

Funcţia *getfillpattern* determină şablonul definit de utilizator cu ajutorul funcţiei *setfillpattern*. Ea are prototipul:

void far getfillpattern(char far ***sablon**);

La apelul funcţiei *getfillpattern* se atribuie, la 8 octeţi din zona spre care pointează pointerul *sablon*, valorile care stabilesc haşura definită de utilizator prin apelul prealabil al funcţiei *setfillpattern*.

Utilizatorul poate trasa şi alte figuri decît cele pentru care au fost prevăzute funcţii standard de felul celor indicate mai sus. În acest scop se pot folosi funcţiile *putpixel, line, lineto, linerel, arc* etc., pentru a trasa elemente componente ale figurii de trasat.

Pentru a colora un domeniu închis obţinut în acest fel,se foloseşte funcţia *floodfill*. Aceasta are prototipul:

void far floodfill(int *x*, **int** *y*, **int** *culoare_frontiera*);

unde:

(x,y) - Sînt coordonatele unui punct interior figurii care se colorează.

culoare_frontiera - Defineşte culoarea utilizată la trasarea conturului figurii.

Interiorul figurii se colorează în conformitate cu parametrii curent setaţi prin apelul funcţiei *setfillstyle*.

La trasarea figurilor trebuie să se ţină seama de faptul că pixelii nu sînt puncte ideale. Ei au forme dreptunghiulare. Din această cauză, figurile afişate pe ecran vor avea un aspect deformat faţă de forma lor ideală. Pentru

a evita acest lucru se utilizează nişte factori de corecţie numiţi coeficienţi de *aspect*. Aceştia sînt doi: unul realativ la abscisă şi unul relativ la ordonată. Valorile lor se pot seta cu ajutorul funcţiei *setaspectratio* de prototip:

void far setaspectratio(int *xaspect*, **int** *yaspect*);

Valorile acestor coeficienţi depind de adaptorul grafic.

Raportul *(double)xaspect/yaspect* se foloseşte adesea la trasarea curbelor definite analitic.

Valorile curente ale acestor coeficienţi pot fi găsite apelînd funcţia *getaspectratio* de prototip:

void far getaspectratio(int far **xaspect*, **int far** **yaspect*);

După apel, cei doi coeficienţi de aspect curenţi se află memoraţi în zonele spre care pointează parametrii *xaspect* şi *yaspect*.

Exerciţii:

19.10 Să se scrie un program care trasează linii orizontale folosind cele 4 stiluri standard şi ambele grosimi.

PROGRAMUL BXIX10

```
#include <graphics.h>
#include <conio.h>

main() /* traseaza linii orizontale folosind cele 4 stiluri standard si
          ambele grosimi */
{
 char *stil[]={
     "SOLID_LINE = 0",
     "DOTTED_LINE =1",
     "CENTER_LINE =2",
     "DASHED_LINE =3"
 };
 char *grosime[]={
     "NORM_WIDTH =1",
     "THICK_WIDTH =3"
 };
 int gd=DETECT,gm;
 int i,j;
 int x,y;

 initgraph(&gd,&gm,"c:\\borlandc\\bgi");
```

```
  x=getmaxx()/4;
  y=getmaxy()/8;
  settextstyle(DEFAULT_FONT,HORIZ_DIR,1);
  settextjustify(LEFT_TEXT,CENTER_TEXT);
  for(i=SOLID_LINE;i<=DASHED_LINE;i++)
      for(j=0;j<2;j++){
          outtextxy(x,y,stil[i]);
          outtextxy(x+textwidth(stil[i])+4,
                      y,grosime[j]);
          y +=textheight(grosime[j])+4;
          if(j)
             setlinestyle(i,0,3);
          else
             setlinestyle(i,0,1);
          line(x,y,3*x,y);
          y+=20;
      }
  getch();
  closegraph();
}
```

9.11 Să se scrie un program care trasează următoarele figuri:

- cerc;
- elipsă;
- dreptunghi;
- patrulater.

Tipul figurii se stabileşte aleator. Figurile sînt trasate folosind în mod aleator culoarea de desenare.

PROGRAMUL BXIX11

```
#include <graphics.h>
#include <conio.h>
#include <stdlib.h>

main() /* traseaza figuri geometrice in mod aleator */
{
 int gd=DETECT,gm;
 int i;
 int xmax,ymax,x,y;
 int poligon[10];
```

```
initgraph(&gd,&gm,"c:\\borlandc\\bgi");
outtextxy(0,0,"Pentru a termina actionati tasta\
          zero");
setviewport(0,8,getmaxx(),getmaxy(),1);

srand(1993); /* seteaza saminta sirului de numere
                pseudo-aleatoare */
xmax=getmaxx();
ymax=getmaxy();
for (;;){
    setcolor(random(15)+1);
    switch(random(4)){
        case 0: /* cerc */
            x=random(xmax/2)+xmax/4;
            y=random(ymax/2)+ymax/4;
            circle(x,y,random(80)+1);
            break;
        case 1: /* elipsa */
            x=random(xmax/2)+xmax/8;
            y=random(ymax/2)+ymax/8;
            ellipse(x,y,0,359,random(80)+1,
            random(60)+1);
            break;
        case 2: /* dreptunghi */
            x=random(xmax/2)+xmax/10;
            y=random(ymax/2)+ymax/10;
            rectangle(x,y,x+random(xmax/4)+1,
                        y+random(ymax/4)+1);
            break;
        case 3: /* patrulater */
            for(i=0;i<8; i+=2){
                poligon[i]=random(xmax);
                poligon[i+1]=random(ymax);
            }
            poligon[8]=poligon[0];
            poligon[9]=poligon[1];
            drawpoly(5,poligon);
            break;
    } /* sfirsit switch */
    if(getch()=='0')
        break;
}
closegraph();
```

```
}
```

19.12 Să se scrie un program care colorează figurile geometrice trasate în programul precedent.

PROGRAMUL BXIX12

```
#include <graphics.h>
#include <conio.h>
#include <stdlib.h>

main() /* traseaza si coloreaza figuri geometrice in mod aleator */
{
 int gd=DETECT,gm;
 int i;
 int xmax,ymax,x,y;
 int poligon[8];

 initgraph(&gd,&gm,"c:\\borlandc\\bgi");
 outtextxy(0,0,"Pentru a termina actionati tasta\
         zero");
 setviewport(0,8,getmaxx(),getmaxy(),1);

 srand(1993); /* seteaza samînta sirului de numere
                 pseudo-aleatoare */
 xmax=getmaxx();
 ymax=getmaxy();
 setcolor(getmaxcolor());
 for (;;){
    setfillstyle(SOLID_FILL,random(15)+1);
    switch(random(4)){
       case 0: /* cerc */
          x=random(xmax/2)+xmax/4;
          y=random(ymax/2)+ymax/4;
          pieslice(x,y,0,360,random(80)+1);
          break;
       case 1: /* elipsa */
          x=random(xmax/2)+xmax/8;
          y=random(ymax/2)+ymax/8;
          fillellipse(x,y,random(80)+1,
          random(60)+1);
          break;
       case 2: /* dreptunghi */
```

```
            x=random(xmax/2)+xmax/10;
            y=random(ymax/2)+ymax/10;
            bar(x,y,x+random(xmax/4)+1,
                y+random(ymax/4)+1);
            break;
        case 3: /* patrulater */
            for(i=0;i<8; i+=2){
                poligon[i]=random(xmax);
                poligon[i+1]=random(ymax);
            }
            fillpoly(4,poligon);
            break;
    } /* sfirsit switch */
    if(getch()=='0')
        break;
 }
 closegraph();
}
```

19.13 Să se scrie un program care afişează 15 dreptunghiuri colorate, pe trei rînduri, folosind toate cele 15 culori ale paletei diferite de culoarea de fond.

Sub fiecare dreptunghi se listează indicele culorii dreptunghiului, în tabloul care defineşte paleta curentă de culori.

După afişarea celor 15 dreptunghiuri se poate regla monitorul aşa încît fiecare dreptunghi să fie vizibil.

Acest program poate fi utilizat ori de cîte ori se constată că monitorul este dereglat.

PROGRAMUL BXIX13

```
#include <stdio.h>
#include <graphics.h>
#include <stdlib.h>
#include <conio.h>

main() /* afiseaza 15 dreptunghiuri colorate */
{
 int Adaptorgrafic,Modgrafic;
 int lat,inalt,x,y,i,j,culoare;
 char asculoare[5];
```

```
 Adaptorgrafic = DETECT;
 initgraph(&Adaptorgrafic,&Modgrafic,
     "c:\\borlandc\\bgi");
 culoare = 1;
 lat = 78;
 inalt = 64;
 x = lat/2;
 y = inalt/2;
 for(j = 0; j <3; j++ ){
     for( i = 0 ; i < 5 ; i++ ){
         setfillstyle(SOLID_FILL,culoare);
         setcolor(culoare);
         bar(x,y,x+lat,y+inalt);
         itoa(culoare,asculoare,10);
         outtextxy(x+lat/2,y+inalt+4,asculoare);
         ++culoare;
         x += (lat/2)*3;
     }
     y += (inalt/2)*3;
     x = lat/2;
 }
 getch();
 closegraph();
}
```

19.14 Să se scrie un program care afişează dreptunghiuri utilizînd toate haşurile standard.

PROGRAMUL BXIX14

```
#include <stdio.h>
#include <graphics.h>
#include <stdlib.h>
#include <conio.h>

main() /* afiseaza dreptunghiuri cu toate hasurile standard */
{
 int Adaptorgrafic,Modgrafic;
 int lat,inalt,x,y,i,j,hasura;
 char ashas[5];

 Adaptorgrafic = DETECT;
 initgraph(&Adaptorgrafic,&Modgrafic,
```

```
                    "c:\\borlandc\\bgi");
    lat = 78;
    inalt = 64;
    x = lat/2;
    y = inalt/2;
    for(j = 0,hasura=EMPTY_FILL; j <3; j++ ){
        for( i = 0 ; i < 5 ; i++ ){
            if(hasura >=12)
                break; /* nu mai sînt hasuri */
            setfillstyle(hasura,WHITE);
            bar(x,y,x+lat,y+inalt);
            rectangle(x,y,x+lat,y+inalt);
            itoa(hasura,ashas,10);
            outtextxy(x+lat/2,y+inalt+4,ashas);
            ++hasura;
            x += (lat/2)*3;
        }
        if(hasura >= 12)
            break;
        y += (inalt/2)*3;
        x = lat/2;
    }
    getch();
    closegraph();
}
```

19.15 Să se scrie un program care trasează un cerc folosind ecuaţiile parametrice ale cercului:

(1) x = r*cos(g) + xcentru;

y = r*sin(g) + ycentru;

unde:

(xcentru,ycentru) - Sînt coordonatele centrului cercului.

r - Este raza cercului.

g - Este unghiul în radiani dintre axa Ox şi razaOM, O fiind originea axelor de coordonate, iar M este punctul de pe cerc de coordonate (x,y).

Cînd unghiul *g* variază de la 0 la 2*PI, punctul M(x,y) descrie cercul de rază *r* şi cu centru în punctul de coordonate (xcentru,ycentru).

Programul afişează două curbe pentru a pune în evidenţă importanţa coeficienţilor de aspect. Prima curbă se afişează folosind ecuaţiile (1), iar

cea de a doua curbă se trasează corectînd ordonata conform relaţiilor de mai jos:

(2) x = r*cos(g) + xcentru;

y = r*(xaspect/yaspect)*sin(g) + ycentru;

unde:

xaspect şi yaspect	- Sînt coficienţii de aspect obţinuţi prin apelul funcţiei *getaspectratio*.

PROGRAMUL BXIX15

```
#include <stdio.h>
#include <graphics.h>
#include <stdlib.h>
#include <conio.h>
#include <math.h>

#define PI 3.14159265

main()
{
 int gd=DETECT,gm;
 int xaspect,yaspect;
 double FactorAspect,draza,rad;
 int x,y,g;
 int xcentru,ycentru;

 initgraph(&gd,&gm,"c:\\borlandc\\bgi");
 getaspectratio(&xaspect,&yaspect);
 FactorAspect=(double)xaspect/yaspect;
 draza=150;
 FactorAspect *=draza;
 xcentru=getmaxx()/2;
 ycentru=getmaxy()/2;

/* traseaza un cerc fara a tine seama de coeficientii de aspect */
 for(g=0;g<360;g++){
     rad=g*PI/180.0;
     x=draza*cos(rad)+xcentru;
     y=draza*sin(rad)+ycentru;
     putpixel(x,y,WHITE);
```

```
  }

  getch();

/* traseaza un cerc tinind seama de coeficientii de aspect */
  for(g=0;g<360;g++){
      rad=g*PI/180.0;
      x=draza*cos(rad)+xcentru;
      y=FactorAspect*sin(rad)+ycentru;
      putpixel(x,y,LIGHTBLUE);
  }
  getch();
  closegraph();
}
```

19.16 Să se scrie un program care deplasează o imagine pe ecran. Acest program a fost propus şi realizat de Negrescu Dan şi Filimon Ovidiu. Programul a apărut şi în cărţile [13] şi [21].

PROGRAMUL BXIX16

```
#include <graphics.h>
#include <conio.h>
#include <alloc.h>
#include <dos.h>

main() /* deplaseaza un vapor pe ecran */
{
 void desen_vapor() ;
 int  grmode,grdrive ;
 void *ptr ;
 void *vapor1,*vapor2 ;
 int  linia,coloana ;

 grdrive=DETECT ;
 initgraph(&grdrive,&grmode,"c:\\borlandc\\bgi") ;
 grmode=EGAHI ;
 setgraphmode(grmode) ;
 setbkcolor(BLACK) ;
 cleardevice() ;
 desen_vapor() ;
 setfillstyle(SOLID_FILL,LIGHTRED) ;
 bar(10,0,25,10) ;
```

```
 vapor1 = malloc(imagesize(0,0,50,50)) ;
 getimage(0,0,50,50,vapor1) ;
 cleardevice() ;
 desen_vapor() ;
 setfillstyle(SOLID_FILL,YELLOW) ;
 bar(25,0,40,10) ;
 vapor2 = malloc(imagesize(0,0,50,50)) ;
 getimage(0,0,50,50,vapor2) ;
 setbkcolor(LIGHTBLUE) ;
 cleardevice() ;
 linia=coloana=0;
 for(;;){
     putimage(coloana,linia,vapor1,XOR_PUT) ;
     delay(100) ;
     putimage(coloana,linia,vapor1,XOR_PUT) ;
     coloana+=25 ;
     if(coloana<570){
         putimage(coloana,linia,vapor2,XOR_PUT) ;
         delay(100) ;
         putimage(coloana,linia,vapor2,XOR_PUT) ;
         coloana +=25 ;
         if(coloana>570)
             coloana=0 ;
     }
     else{
         linia+=60 ;
         coloana=0 ;
         if(linia>300)
             linia=0 ;
     }
     if(kbhit())
         break ;
 }
 getch() ;
 closegraph() ;
 free(vapor1) ;
 free(vapor2) ;
}

void desen_vapor() /* deseneaza un vapor */
{
 int puncte[]={
      25,10,
```

```
        10,30,
         0,30,
        10,50,
        40,50,
        50,30,
        40,30,
        25,10
  } ;

  setcolor(CYAN) ;
  drawpoly(8,puncte) ;
  line(10,30,40,30) ;
  setfillstyle(SOLID_FILL,LIGHTMAGENTA) ;
  floodfill(25,45,CYAN) ;
  setfillstyle(SOLID_FILL,WHITE) ;
  floodfill(25,20,CYAN) ;
  setcolor(BROWN) ;
  line(25,10,25,30) ;
}
```

BIBLIOGRAFIE

1. **C.Bohn, G.Jacopini**

   FlowDiagrams, Turning Machines and Languages with OnlyTwo Formation Rules

   Comm ACM, 9, 5, pag. 366 - 371, 1966

2. **Brian W. Kernighan, Dennis M. Ritchie**

   The C Programming Language

   Prentice-Hall, Inc, Englewood Cliffs

   New Jersey 1978

3. **Donald E. Knuth**

   Tratat de programarea calculatoarelor, vol. 1 - 3

   Editura Tehnică

   Bucureşti 1974

4. **Valentin Cristea şi alţii**

   Dicţionar de informatică

   Editura ştiinţifică şi enciclopedică

   Bucureşti 1981

5. **Narian Gehani**

   C: An Advanced Introduction

   AT&T Bell Laboratories Murray Hill

   New Jersey 1985

6. **Joel E. Richardson, Michel J. Carey, Daniel T. Schuh**

   The Design of the E Programming Language

   Computer Sciences Department

   University of Wisconsin Madison

Wi53706 1989

7. * * *

BORLAND INTERNATIONAL TURBO C++:

Getting Started

1990

8. * * *

BORLAND INTERNATIONAL TURBO C++:

Programmer's Guide

1990

9. **Frederic Lung**

Le Langage C++

CNIT Paris - La Defense

1990

10. **Setrag Khoshafian, Razmik Abnous**

Object Orientation Concepts, Languages, Databases, User Interfaces

John Wiley&Sons, Inc. New York

Chichester Brisbone Toronto

Singapore 1990

11. **Aaron M. Tenenbaum, Yedidyah Langsam, Moshe J. Augenstein**

Data Structures Using C

Prentice - Hall International, Inc.

1990

12. **Clara Ionescu, Ioan Zsako**

Structuri arborescente cu aplicaţiile lor

Editura Tehnică

Bucureşti 1990

13. **Liviu Negrescu**
Introducere în limbajul C vol. 1 - 2
Colecţia GLOB
1990

14. **Stroustrup Bjarne**
The C++ Programming Language
Second Edition
Copyright 1991 by AT&T
Bell Telephone Laboratories, Incorporated

15. **Liviu Negrescu**
Limbajul C - Culegere de programe
Fasciculele 1 - 2
Cluj-Napoca 1991

16. **Liviu Negrescu**
Limbajul TURBO C
Editura LIBRIS
Cluj-Napoca 1992

17. **Liviu Negrescu**
Introducere în mediul de dezvoltare integrat TURBO C
Editura LIBRIS
Cluj-Napoca 1992

18. **Ionuţ Muşlea**
Iniţiere în C++ - Programare orientată pe obiecte
Editura MicroInformatica

Cluj-Napoca 1992

19. **Donald L. Shell**
CACM 2(iulie), pag. 30-32
1959

20. **C. A. R. Hoare**
Quicksort
Comp. j., 5, No. 1
1962

21. **Liviu Negrescu**
Introducere în limbajul C
Editura MicroInformatica
Cluj-Napoca 1993



str. Observatorului nr.1, bl.OS1
3400 Cluj-Napoca
of. PTTR Cluj-Napoca, CP 186
tel / fax 064.198263, 123125

Cartea se adresează unui cerc larg de cititori care doresc să se inițieze în programarea și utilizarea limbajelor C și C++.

Primele două volume realizează o introducere elementară în aceste limbaje.

Volumul trei conține programe diverse în C și C++ cu aplicații la rezolvarea problemelor științifice și tehnico-inginerești, în prelucrări de date, precum și în scrierea programelor de sistem.



I.S.B.N. 973-96980-9-3

14950 lei